НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Інститут прикладного системного аналізу

Кафедра математичних методів системного аналізу

До захисту допущено:

Завідувач кафедри

_____ Оксана ТИМОЩУК

«___»_____20__ р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Системи та методи штучного інтелекту» спеціальності 122 «Комп'ютерні науки та інформаційні технології»

на тему: «Система відновлення динаміки часового ряду методом штучних нейромереж»

Виконав:
студент IV курсу, групи КА-65
Андросов Дмитро Васильович _____

Керівник:
доцент, д.т.н. Недашківська Надія Іванівна _____

Консультант з економічної частини:
доцент, к.е.н. Шевчук Олена Анатоліївна _____

Консультант з нормоконтролю:
доцент, к.т.н. Коваленко Анатолій Єпіфанович _____

Рецензент:
доцент, к.т.н. Безносик Олександр Юрійович _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2020 рік

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Інститут прикладного системного аналізу
Кафедра математичних методів системного аналізу

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 122 «Комп'ютерні науки та інформаційні технології»
Освітньо-професійна програма «Системи та методи штучного інтелекту»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Оксана ТИМОЩУК
«25» травня 2020 р.

ЗАВДАННЯ
на дипломну роботу студенту
Андросову Дмитру Васильовичу

1. Тема роботи «Система відновлення динаміки часового ряду методом штучних нейромереж», керівник роботи Недашківська Надія Іванівна, доцент, д.т.н., затверджені наказом по університету від «25» травня 2020 р. № 1143-с

2. Термін подання студентом роботи 8.06.2020.

3. Вихідні дані до роботи результати прогнозу нейронною мережею штучно згенерованого нестаціонарного часового ряду.

4. Зміст роботи аналіз існуючих підходів моделювання часових рядів, визначення архітектури моделі часового ряду на основі штучних нейронних мереж, реалізація процесу навчання мережі, проведення порівняльного аналізу з використанням різних початкових умов постановки експерименту, інтерпретація отриманих результатів.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо) зображення вхідних даних, архітектур моделей, результатів прогнозування та моделювання, презентація до виступу.

6. Консультанти розділів роботи*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Шевчук О.А., доцент		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Формування теми БДР	03.09.2019	Виконано
2.	Огляд літератури за тематикою роботи та її опрацювання	03.09.2019-01.03.2020	Виконано
3.	Програмна реалізація моделі	11.03.2020-20.04.2020	Виконано
4.	Написання першого розділу БДР	20.04.2020-27.04.2020	Виконано
5.	Узгодження теми БДР з керівником	28.04.2020	Виконано
6.	Написання другого розділу БДР	01.05.2020-07.05.2020	Виконано
7.	Написання третього розділу БДР	14.05.2020-27.05.2020	Виконано
8.	Написання четвертого розділу БДР	27.05.2020-29.05.2020	Виконано

Студент Дмитро АНДРОСОВ

Керівник Надія НЕДАШКІВСЬКА

* Якщо визначені консультанти. Консультантом не може бути зазначено керівника дипломної роботи.

РЕФЕРАТ

Дана дипломна робота містить 83 ст., 4 ч., 7 табл., 16 рис., 2 дод., 15 джерел.

АНАЛІЗ ЧАСОВИХ РЯДІВ, ШТУЧНІ НЕЙРОННІ МЕРЕЖІ, РЕКУРЕНТНІ НЕЙРОННІ МЕРЕЖІ, ЗВИЧАЙНІ ДИФЕРЕНЦІАЛЬНІ РІВНЯННЯ, АРХІТЕКТУРА КОДУВАЛЬНИК-ДЕКОДУВАЛЬНИК

Об’єкт дослідження – процес відновлення динаміки часового ряду.

Мета роботи – розглянути теоретичні основи моделювання та прогнозування часових рядів, засвоїти принципи моделювання часових рядів методами штучних нейронних мереж, розробити модель часового ряду на основі штучних нейронних мереж.

Методи дослідження – аналіз процесу створення моделі часового ряду, експеримент, результатом якого є програмна реалізація моделі часового ряду, аналіз отриманих результатів.

Результатом роботи є модель часового ряду, а саме система відновлення динаміки часового ряду методами штучних нейронних мереж.

Новизною роботи є створення способу відновлення динаміки часового ряду на основі інтеграції апарату диференціальних рівнянь до шарів штучних нейронних мереж, а також використання підходу моделей кодувальник-декодувальник для виявлення прихованих закономірностей у даних, що досліджуються.

Результати даної роботи можна застосовувати для створення гібридних моделей прогнозування різноманітних процесів. Також, розроблена реалізація моделі є невимогливою до характеристик досліджуваного процесу, таких як нестаціонарність, сезонність, тощо.

ABSTRACT

This thesis contains 83 pages, 4 sections, 7 tables, 16 figures, 2 aappendices, 15 sources.

TIME SERIES ANALYSIS, ARTIFICIAL NEURAL NETWORKS, NEURAL ORDINARY DIFFERENTIAL EQUATIONS, RECURRENT NEURAL NETWORKS, VARIATIONAL AUTOENCODERS

This study covers the time series' dynamics reconstruction process.

The main aim of this paper is to review the foundations of time series analysis using deep learning techniques and approaches, and to develop a neural network based time series model.

Research method, used in this study, is conducting an experiment that leads to developing a time-dependent process predictive model.

The result of this work is a time series forecasting pipeline based on neural networks approach.

Novelty of this study is in presenting a new family of artificial neural networks and their applications in solving cases that involves dynamics reconstruction of continuous-time-dependent processes and their combination with modern feature extraction architectures, such as encoder-decoder models.

Results of this research are related to developing hybrid models for analysis and forecasting different processes.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ ТА СКОРОЧЕНЬ

ЗДР – звичайне диференціальне рівняння

МНК – метод найменших квадратів

РК – метод Рунге-Кутти

РМНК – рекурсивний метод найменших квадратів

РНМ – рекурентна нейронна мережа

ШНМ – штучна нейронна мережа

ARIMA – Auto Regression Integrated Moving Average

ARMA – Auto Regression Moving Average

GARCH – Generalized Auto Regression Conditional Heteroscedasticity

GRU – Gated Recurrent Unit

KL-divergence – Kullback-Leibler divergence

LODE – Latent Ordinary Differential Equations

LSTM – Long Short-Term Memory

VAE – Variational Autoencoder

ВСТУП

Аналіз часових рядів є одним із найпопулярніших розділів прикладної статистики. Дійсно, робота з часовими рядами є неминучою, коли мова йде про планування та прогнозування різноманітних процесів в економіці та торгівельній справі на основі історичних даних.

В наш час, різними компаніями та підприємствами створюються надвеликі сховища історичних даних для створення ефективних моделей прогнозування внутрішніх та зовнішніх процесів.

Дисципліна аналізу часових рядів за час свого розвитку створила такі популярні моделі-фреймворки, як Авторегресія з Ковзним Середнім (ARMA-моделі), Авторегресія з Інтегрованим Ковзним Середнім (ARIMA-моделі), та інші.

Наведені вище моделі найкраще працюють зі стаціонарними рядами, тобто такими, що не змінюють свої характеристики (наприклад, середнє значення та середньоквадратичне відхилення) з часом. Але зрозуміло, що більшість часових рядів, що спостерігаються у реальному світі, неможливо віднести до таких.

Для того, щоб привести ряд до стаціонарного, виконуються необхідні маніпуляції, наприклад виявлення тренду ряду і приведення ряду до вигляду ряду з константним трендом (що виконується для побудови ARIMA-моделей), моделювання умовної гетероскедасичності ряду (GARCH-моделі), або використання гібридних моделей (ARIMA/GRCH для моделювання середнього та дисперсії досліджуваного ряду).

Зрозуміло, що обробка даних є одним із найбільш тривалих та громіздких процесів у процесі побудови адекватних моделей прогнозування, тому проблема автоматизації даного процесу постає однією з ключових проблем побудови адекватних прогнозних моделей.

Стрімкий розвиток дисципліни машинного навчання та поява моделей глибинного навчання зробили можливим часткову автоматизацію обробки даних за рахунок можливостей побудови високорівневих абстракцій.

Розвиток моделей глибинного навчання дозволяють виявляти закономірності в структурованих у часі даних для точного прогнозування наступних значень варіант на основі історичних даних. Існуючі рішення та підходи дозволяють відтворювати вплив прихованих факторів для побудови адекватних дескриптивних моделей.

Такі нові рішення як рекурентні мережі, що здатні встановлювати закономірності у даних на основі попередніх спостережень, автокодувальники, що здатні виокремлювати найважливіші ознаки у даних, можуть бути використаними і при побудові моделей часових рядів.

Користуючись штучними нейромережами як універсальними апроксиматорами функцій, можна відтворювати динаміку досліджуваних процесів, моделюючи не сам дискретний часовий ряд, а переходячи до аналогу даного ряду, що залежить від неперервного часу.

Таким чином, з метою забезпечення автоматизації обробки даних та побудови адекватних моделей прогнозування нестаціонарних часових рядів була створена модель прогнозування динаміки часового ряду на основі звичайних диференціальних рівнянь.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз існуючих моделей часових рядів

1.1.1 Авторегресія з ковзним середнім (ARMA)

Нагадаємо, що часовий ряд

$$Y = \{y_t, t \in \mathbb{Z}\}$$

є ARMA(p, q) процесом, якщо Y – стаціонарний процес, та при цьому виконується:

$$y_t = \theta_0 + \sum_{i=1}^{p} \theta_i y_{t-i} + \sum_{j=1}^{q} \gamma_j \epsilon_{t-j} \quad (1.1)$$

де $\{\epsilon_t\} \sim N(0, \sigma^2)$;

$\{\theta_i\} \in \mathbb{R}$;

$\{\gamma_i\} \in \mathbb{R}$ [1].

Коефіцієнти даної моделі зазвичай знаходять методом найменших квадратів (МНК) або рекурсивним МНК.

Оскільки рівняння (1.1) містить компоненти білого шуму, то для прогнозування на довільну кількість кроків s можна скористатися формулою прогнозування середнього значення $\hat{y}_{t+s}$:

$$\hat{y}_{t+s} = \theta_0 + \sum_{i=1}^{p} \theta_i \hat{y}_{t+s-i} \quad (1.2)$$

Також, оскільки процес, що необхідно моделювати, є стаціонарним, то похибка прогнозування, яку отримано шляхом віднімання (1.2) від (1.1), рівна:

$$y_{t+s} - \hat{y}_{t+s} = err_s \sim N(0,\ \sigma^2(\sum_{r=1}^{s}(\sum_{i}^{q}{\gamma_i}^2(1+{\theta_r}^2) + \sum_{j=1}^{r}{\gamma_{q+j}}^2))$$

[2]. Тобто дисперсія похибки значно зростає при збільшенні кроку прогнозу і асимптотично збігається тільки якщо $\{{\theta_r}^2\} < 1, \{{\gamma_j}^2\} < 1$.

1.1.2 Авторегресія з інтегрованим ковзним середнім (ARIMA)

ARIMA-моделі відрізняються від розглянутих у попередньому розділі тим, що не потребують стаціонарності спостережуваного процесу, а приводить його до стаціонарного шляхом оператора кінцевих різниць (диференціювання), позначим його через ∇^d, d – порядок диференціювання. Таким чином, часовий ряд

$$Y = \{y_t, t \in \mathbb{Z}\}$$

є ARIMA(p, d, q) процесом, якщо $\nabla^d Y$ – стаціонарний процес, тобто ARIMA(p, d, q) = ARMA(p+d, q) [1].

1.1.3 Узагальнена умовна гетероскедастичність (GARCH)

Дана модель застосовується також до стаціонарних процесів. Тобто, процес

$$Y = \{y_t, t \in \mathbb{Z}\}$$

є GARCH(q, p) процесом, якщо Y – стаціонарний процес і при цьому виконується:

$$y_t = \epsilon \sqrt{\alpha + \sum_{i=1}^{p} \alpha_i {y_{t-i}}^2 + \sum_{j=1}^{q} \beta_j {\sigma_{t-j}}^2} \quad (1.3)$$

де $\epsilon \sim N(0, {\sigma_\epsilon}^2)$;

$\{\alpha_i\} \in \mathbb{R}^+$;

$\{\beta_i\} \in \mathbb{R}^+$ [1].

В даному випадку існує потреба стаціонарності ряду, щоб безумовна дисперсія процесу, що описується (1.3) була рівною

$$\sigma^2 = \frac{{\sigma_\epsilon}^2}{1 - \sum_i \alpha_i - \sum_j \beta_j}$$

[2].

1.2 Аналіз методів машинного навчання для обробки та прогнозування послідовностей. Прогнозування часового ряду як задача навчання з вчителем

1.2.1 Рекурентні нейронні мережі (Simple RNN)

Нехай матимемо справу з часовими рядами однієї змінної, тобто

$$Y = \{y(t), t \in \mathbb{Z}\}$$

Тоді можна скласти такий навчальний набір

$$D = \left\{d_i \mid d_i = \langle \overrightarrow{X_i}, y_{t-i} \rangle\right\}$$

$$\overrightarrow{X_i} = \left\{y_{t-i-1}, y_{t-i-2}, \dots, y_{t-i-p}\right\}$$

Таким чином, було задачу навчання з учителем, сутність якої полягає у тому, щоб за наявності значень p лагів даної послідовності зробити прогноз на 1 крок. Таку задачу можна розв'язати методами машинного навчання. Одним з найбільш ефективних фреймворків для задач прогнозування та обробки послідовностей є Рекурентні Нейронні Мережі (RNN).

Наведемо нижче архітектуру такої мережі (рис. 1.1.).

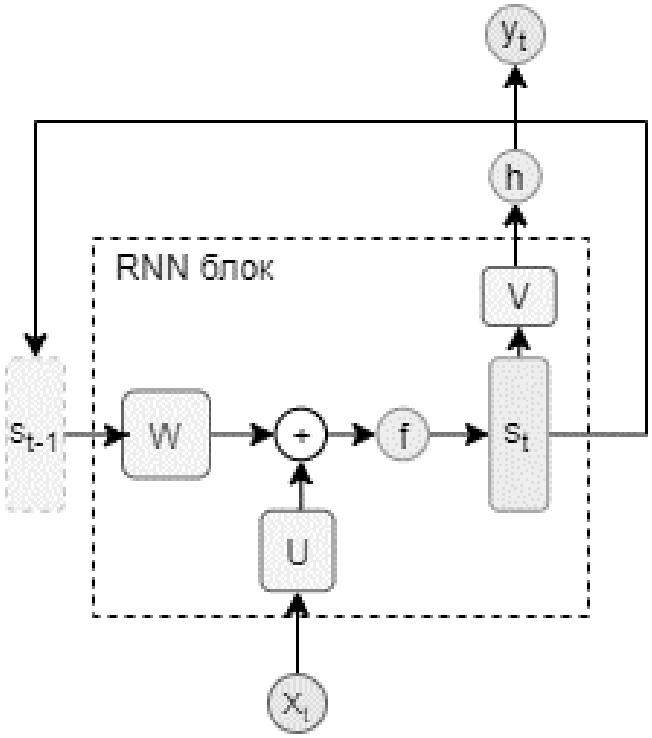


Рисунок 1.2 – Архітектура RNN мережі

Позначимо через W матрицю ваг, що слугує для переходу між прихованими станами мережі, через U матрицю ваг для вхідного вектору, а через V- для виходів. Тоді, отримуємо таке формальне визначення рекурентної мережі:

$$a_\tau = b + Ws_{\tau-1} + Ux_\tau, \tag{1.4}$$

$$o_\tau = c + Vs_\tau, \tag{1.5}$$

$$s_\tau = f(a_\tau), \tag{1.6}$$

$$y_\tau = h(o_\tau), \tag{1.7}$$

де b, с - вектори зміщення;

f, h - функції активації вхідного та вихідного шарів відповідно [3].

Як бачимо з системи (1.4)-(1.7) дані мережі мають цікаву особливість, яка полягає у тому, що на відміну від мереж прямого розповсюдження (наприклад, багатошаровий перцептрон), у рекурентних мереж зв'язки між нейронами можуть йти не тільки послідовно від попереднього шару до наступного, а й «до самого себе», точніше, до свого стану у попередній момент часу.

Саме таке «зациклювання» графу обчислень і дозволяє моделювати залежності поточних компонент від попередніх, тому що якщо ми будемо використовувати звичайні мережі прямого поширення, то кожна компонента кортежу даних буде вважатися незалежною від інших, що є некоректним припущенням для задачі обробки часових даних.

Тобто, в силу «зациклення» зав'язків рекурентного блоку відносно часу та особливості параметрів (ваг) кожного блоку (ваги інваріантні від часу та спільні для самого блоку) мережа може вивчити залежність кожної змінної від довільної кількості її попередніх значень.

Тепер нам необхідно обрати функцію похибок для подальшої оптимізації параметрів мережі. Оскільки необхідно моделювати послідовності, то в якості функції похибок для даної мережі можна застосувати від'ємну логарифмічну правдоподібність (neg-log-likelihood), яку будемо мінімізувати, тобто:

$$L = \sum_{i} L_{t=i} = -\sum_{i=0}^{t-p} \log\left(p\left(y_{t-i} | y_{t-i-1}, y_{t-i-2}, \dots, y_{t-i-p}, \vec{\theta}\right)\right), \tag{1.8}$$

$$\hat{y}_{t-i} = arg \min_{\vec{\theta}} L_i(\vec{\theta}) \tag{1.9}$$

де $\vec{\theta} = \langle W, V, U, b, c, o, s \rangle$.

Сконцентруємося на тому, як тепер таку мережу навчати. Хоча вони й відрізняються від звичайних мереж лише одним елементом у графі обчислень,

та ця відмінність уже не дозволяє нам використати алгоритм зворотного поширення похибки без низки модифікацій.

У мережах прямого поширення похибка обчислювалась за правилом диференціювання складної функціїї як похідна всіх композицій вихідних функцій шарів, що цей нейрон використовують. Але невідомо як диференціювати, коли граф обчислень не є ациклічним, тому треба використовувати більш хитрі механізми.

Насправді, при розгортці графу обчислень відносно часу будь-яка ациклічність зникає, тому будемо навчати мережу не «ітеративно» (від епохи до епохи), а рекурентно відносно часу.

Метод, що дозволяє навчити мережу таким чином, називається методом зворотного поширення похибок у часі (backpropagation through time).

Користуючись цим методом, ми рухаємось назад у часі від кінцевої варіанти ряду до початкової. Наведемо загальні формули для обчислення градієнтів по параметрам моделі:

$$\frac{\partial L}{\partial L_i} = 1 \; \forall i, \tag{1.10}$$

$$(\nabla_{o_\tau} L)_i = \frac{\partial L}{\partial L_i}\frac{\partial L_i}{\partial o_{\tau_i}} = \frac{\partial L_i}{\partial o_{\tau_i}} \; \forall i \; \forall \tau \tag{1.11}$$

Оскільки прихований стан s_τ в кінцевий момент часу $\tau = t - i - p$ має лише одного нащадка у графі обчислень - o_τ, то:

$$\nabla_{s_\tau} L = V^T \nabla_{o_\tau} L \tag{1.12}$$

Далі виконуючи ітерації обернено часу, отримуємо

$$\nabla_{s_\tau} L = \left(\frac{\partial s_{\tau+1}}{\partial s_\tau}\right)^T \left(\nabla_{s_{\tau+1}} L\right) + \left(\frac{\partial o_\tau}{\partial s_\tau}\right)^T \left(\nabla_{o_\tau} L\right), \tag{1.13}$$

$$\nabla_c L = \sum_\tau \left(\frac{\partial o_\tau}{\partial c}\right)^T \nabla_{o_\tau} L = \sum_\tau \nabla_{o_\tau} L, \tag{1.14}$$

$$\nabla_b L = \sum_\tau \left(\frac{\partial s_\tau}{\partial b}\right)^T \nabla_{s_\tau} L, \tag{1.15}$$

$$\nabla_V L = \sum_i \sum_\tau (\nabla_{o_\tau} L)_i \nabla_V o_{\tau_i} = \sum_\tau (\nabla_{o_\tau} L)^T s_\tau, \tag{1.16}$$

$$\nabla_W L = \sum_i \sum_\tau (\nabla_{s_\tau} L)_i \nabla_W s_{\tau_i}, \tag{1.17}$$

$$\nabla_U L = \sum_i \sum_\tau (\nabla_{s_\tau} L)_i \nabla_U s_{\tau_i} \tag{1.18}$$

Формули (1.10)-(1.18) для задачі класифікації наведені у [4].

Далі, використаємо знову позначення $\vec{\theta} = \langle W, V, U, b, c, o, s \rangle$ для спрощення викладення алгоритму зворотного поширення похибки. Нехай кожний параметр позначимо через $\theta_{q\mathcal{J}_q}$, $q = \overline{1,7}$, $\mathcal{J}_q$ – множина індексів компоненти q вектору $\vec{\theta}$. Введемо також позначення $\Delta_{min}, \Delta_{max}$ для мінімального та максимально значень оновлення значень параметрів моделі відповідно.

Далі перевіряємо характер зміни знаку градієнтів функції похибки по параметрам від ітерації $(k-1)$ до ітерації (k). Маємо три випадки:

а) Нехай

$$\frac{\partial L^{(k-1)}}{\partial \theta_{q\mathcal{J}_q}} \frac{\partial L^{(k)}}{\partial \theta_{q\mathcal{J}_q}} > 0$$

тоді

$$\Delta_{q\mathcal{J}_q}^{(k+1)} = \min(\mu_+ \Delta_{q\mathcal{J}_q}^{(k)}, \Delta_{max})$$

б) Нехай

$$\frac{\partial L^{(k-1)}}{\partial \theta_{q\mathcal{J}_q}} \frac{\partial L^{(k)}}{\partial \theta_{q\mathcal{J}_q}} < 0$$

тоді

$$\Delta_{q\mathcal{J}_q}^{(k+1)} = \max(\mu_- \Delta_{q\mathcal{J}_q}^{(k)}, \Delta_{min})$$

в) Якщо

$$\frac{\partial L^{(k)}}{\partial \theta_{q\mathcal{J}_q}} = 0$$

тоді

$$\Delta_{q\mathcal{J}_q}^{(k+1)} = \Delta_{q\mathcal{J}_q}^{(k)}$$

У формулах вище μ_+, μ_- - додатні константи зміни швидкості навчання.

Після того, як було отримано нову величину оновлення параметрів, визначаємо нові значення параметрів:

$$\theta_{q\mathcal{J}_q}^{(k+1)} = \theta_{q\mathcal{J}_q}^{(k)} - sign(\frac{\partial L^{(k)}}{\partial \theta_{q\mathcal{J}_q}})\Delta_{q\mathcal{J}_q}^{(k+1)} \quad (1.19)$$

Таким чином, отримано алгоритм Rprop [5].

Оскільки оновлення параметрів проходитимуть не лише ітераційно по відношенню до пакетів даних, а й відносно прихованих станів у часі, то використовуючи такі функції активації як гіперболічний тангенс або сигмоїдну функцію, отримаємо, що оновлення ваг на найбільш «віддалених» відносно часу ітерацій буде близьким до нуля.

Для задач прогнозування рядів, кожне наступна варіанта якого залежить від невеликої кількості значень попередніх варіант, дані мережі є ефективними. Але якщо ми будемо моделювати ряд, наприклад, з річною сезонністю, то проблема експоненційно затухаючого градієнту не дасть нам змоги реконструювати залежності такого типу і навряд чи допоможе у прогнозуванні.

Також існує проблема «вибуху» градієнта (exploding gradient), що зумовлюється рекурсією відносно часу, рівнянням оновлення ваг (1.19) та спільним набором параметрів $\vec{\theta}$ для рекурентного блоку та великими абсолютними значеннями цих параметрів. Тобто, при зворотному поширенні

у часі градієнту через такі параметри значення градієнтів експоненційно зростають [3].

1.2.2 Довга короткострокова пам'ять (LSTM)

Як було обговорено раніше, звичайні RNN через існування проблеми затухаючих градієнтів дуже погано справляються з довгостроковими залежностями, що є їх суттєвим недоліком.

Щоб вирішити таку ситуацію, потрібно, в першу чергу модифікувати саму архітектуру мережі, зробити її більш складною, але більш пристосованою до запам’ятовування тривалих залежностей, наприклад річної сезонності.

Виявляється, що якщо збільшити «вкладеність» мережі та замість параметрів-констант використовувати механізм управляючих блоків, які будуть імітувати «пам'ять», слугуючи своєрідним резервуаром найвагоміших значень. Саме шляхом таких модифікацій і були отримані LSTM-мережі.

Дані мережі, як зазначалося раніше, відрізняються від звичайних рекурентних більш складною конструкцією, що дозволяють їм стримувати градієнт похибки від «затухання» та «вибуху». Таким стримуючим механізмом є додатковий управляючий блок, що робить значення ваг кожної петлі рекурентного блоку не константним, а обумовленим від контексту.

Інакше кажучи, замість блоку, що застосовує нелінійності щодо афінних перетворень вхідних даних та рекурентних блоків, LSTM-мережі складаються з блоків, що містять в собі внутрішню рекурсію в доповненні до зовнішньої. На додачу до цього механізму вкладеної рекурсії в кожному блоці є свої управляючі вентильні механізми («гейти»), котрі маніпулюють потоками інформації в кожному конкретному блоці.

Наведемо схематичне зображення структури LSTM блоку нижче (рис. 1.2.):

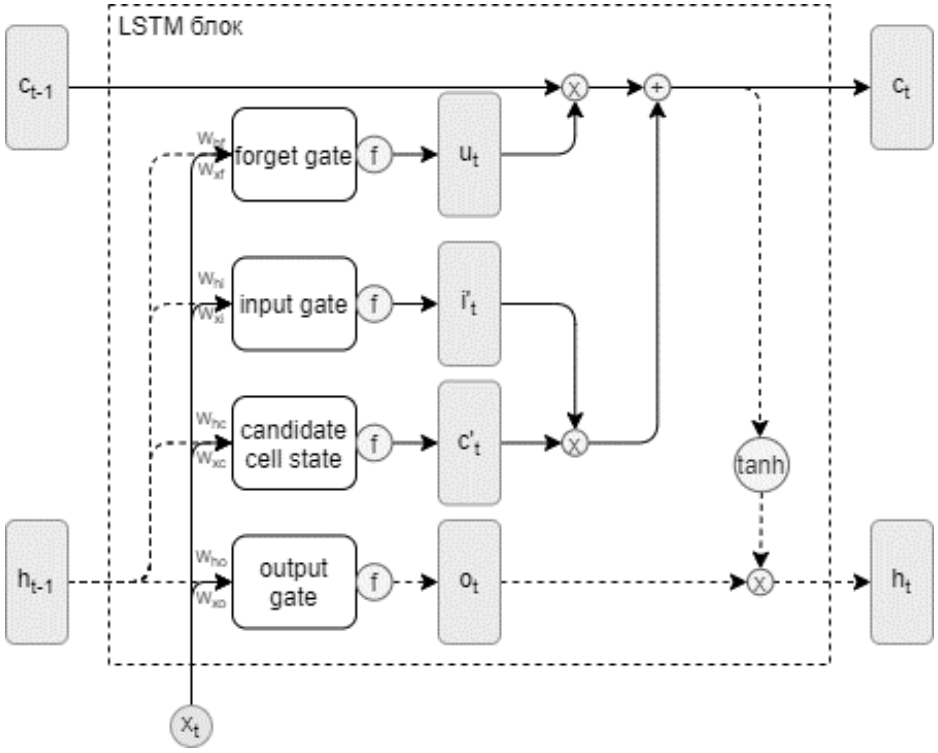


Рисунок 1.2 – Архітектура LSTM мережі

Як бачимо з рисунку, у мережі такого типу є три основних та один додатковий «шлюзи» (або «гейти») – «гейт забування», вхідний, вихідний та «гейт»-кандидат.

Нехай у момент часу τ на вхід до блоку подається вектор x_τ. Опишемо формально систему рівнянь, що задає даний блок, користуючись позначеннями з рисунку вище:

$$c'_t = \tanh(W_{xc}x_\tau + W_{hc}h_{\tau-1} + b_{c'}), \quad (1.20)$$

$$i_\tau = \sigma(W_{xi}x_\tau + W_{hi}h_{\tau-1} + b_i), \quad (1.21)$$

$$f_\tau = \sigma\left(W_{xf}x_\tau + W_{hf}h_{\tau-1} + b_f\right), \quad (1.22)$$

$$o_\tau = \sigma(W_{xo}x_\tau + W_{ho}h_{\tau-1} + b_o), \quad (1.23)$$

$$c_\tau = f_\tau \odot c_{\tau-1} + i_\tau \odot c'_\tau, \quad (1.24)$$

$$h_\tau = o_\tau \odot \tanh(c_\tau) \quad (1.25)$$

Тут і надалі ми позначатимемо через оператор «⊙» поелементне множення матриць. Розглянемо детальніше систему (1.20)-(1.25). Спочатку на

вхід до блоку подається вектор x_τ та вектор прихованого стану за попередній часовий проміжок $h_{\tau-1}$. Далі ми створюємо нову комірку пам'яті (точніше, комірку-кандидат) c'_t для того щоб передати її в комірку пам'яті c_τ. c_τ являє собою лінійну комбінацію «гейта пам'яті» за період $\tau - 1$, помножену на значення «гейта» забування та нового кандидата в «гейт пам'яті», помноженого на вхідний «гейт». За допомогою цих керуючих змінних f_τ та i_τ ми будемо вирішувати, чи слід мережі запам'ятовувати значення нового вхідного вектору, та чи треба «забути» попереднє значення «пам'яті», причому в силу динамічності керуючих змінних значення c_τ може мати різний характер для різних компонент вхідного вектору x_τ.

Також звернемо увагу, що ми використовуємо у частині формул не скалярний добуток, а поелементний. Це пов'язано з тим, щоб не «затерти» всю пам'ять LSTM-блоку, а лише її частина буде оновлена.

Також перевагою такої конструкції є те, що можна не тільки доповнювати «пам'ять» та «перезаписувати» її, а й використати будь-яку лінійну комбінацію значень «пам'яті» за різні часові проміжки.

Така особливість архітектури робить дані мережі дуже гнучкими в плані запам'ятовування закономірностей.

Як бачимо з алгоритму, LSTM-мережі хоча і є більш громіздкими, ніж звичайні RNN, але вони вирішують проблему затухаючих градієнтів, оскільки градієнт відносно «пам'яті»

$$\nabla_{c_\tau} L = \left(\frac{\partial \mathrm{c}_\tau}{\partial \mathrm{c}_{\tau-1}}\right)^T \nabla_{\mathrm{c}_{\tau-1}} L = f_\tau \nabla_{\mathrm{c}_{\tau-1}} L = \prod_i f_i$$

при відсутності значень, близьких до нуля, дозволяє помилці поширюватися (відносно) без затухання упродовж всього «резерву» запам'ятовування. Проблему «вибуху» градієнтів, у свою чергу, можна вирішити методом

«урізання» градієнтів (gradient cropping), тобто виставленням порогового значення, вище якого градієнту присвоюється значення даного порогу [3].

Існує низка модифікацій LSTM архітектури, наведеної на рис. 1.2, що додають (або виключають) нові навчальні параметри (peephole-LSTM, LSTM без одного з основних гейтів, тощо), але всі такі архітектури не перевершують по показникам продуктивності та по ключовим метрикам (наприклад, точності) архітектуру, що задається рівняннями (1.20)-(1.25) [3].

1.2.3 Рекурентні вентильні мережі (GRU)

Як зазначалося у попередньому пункті, LSTM-мережі оперують великою кількістю параметрів та великою вкладеністю моделі. В звичайних РНМ кожний блок керувався одним прихованим вектором, а входи, прихований стан та виходи блоку мали один і по одному набору ваг. В LSTM архітектурі, що освітлена на рис. 1.2., містить одразу вісім матриць ваг. Якщо ми пригадаємо про рекурентність відносно часу, то зрозуміємо, що параметрів дійсно дуже багато.

Як виявилося, критично важливими у архітектурі LSTM є тільки два «гейти» - вхідний та «забування». Крім того, зрозуміло, що ключовим поняттям є сама пам'ять. Однією з найбільш успішних спроб модифікації LSTM є рекурентні вентильні мережі GRU.

Даний тип мереж є певним компромісом між «стандартними» RNN та LSTM. GRU – це модифікація LSTM мережі, що позбавляє останню вхідного «гейту» i_τ і керуючими залишається тільки «гейт» забування f_τ, який у термінології GRU називається «гейтом» оновлення (update gate), будемо позначати його через u_τ. Введемо «гейт» скидання (reset gate), який буде керувати перенесенням «пам’яті» з попереднього кроку на поточний у процесі конструювання кандидата на «гейт пам’яті» до нелінійного перетворення самої «пам’яті», і позначимо його через r_τ.

«Гейти» скидання та оновлення можуть «ігнорувати» частину компонент вектору поточного стану мережі. «Гейт» оновлення у даному випадку може інтерпретуватись як лінійний leaky-інтегратор. Тобто, цей «гейт» може у певний момент часу або скопіювати вхід, або повністю проігнорувати його, в залежності від параметрів моделі. «Гейти» скидання, у свою чергу контролюють, які частини поточного стану необхідно передати далі для обчислення наступного стану блоку та дозволяють виразити нелінійних характер залежності поточного та попереднього станів мережі [4].

Таким чином, рівняння (1.20)-(1.25) набувають наступного вигляду:

$$u_\tau = \sigma(W_{xu}x_\tau + W_{hu}h_{\tau-1} + b_u), \tag{1.26}$$

$$r_\tau = \sigma(W_{xr}x_\tau + W_{hr}h_{\tau-1} + b_r), \tag{1.27}$$

$$h'_\tau = \tanh\big(W_{xh}x_\tau + W_{hh'}(r_\tau \odot h_{\tau-1})\big), \tag{1.28}$$

$$h_\tau = u_t \odot h_{\tau-1} + (1 - u_\tau) \odot h'_\tau \tag{1.29}$$

Також бачимо, що у даних мереж прихований стан та вихідний поєднані між собою.

Аналогічно мереж LSTM, архітектура GRU може вирішити проблему затухання градієнту відносно «пам’яті». Для цього достатньо взяти похідну функції помилок L відносно $h_\tau \ \forall\tau$.

Також слід правильно обрати початковий вектор b_u. Оскільки в моделях машинного навчання на початковій ітерації значення усіх параметрів моделі є випадковими величинами, розподіленими на одиничному інтервалі, такий тип ініціалізації може призвести то експоненційного затухання градієнту навіть за умови близькості u_τ до 1 на перших ітераціях. Малі значення b_u призводять до того, що за певну кількість кроків в силу сигмоїдної функції актвації значення u_τ будуть близькими до $1/2$, що в силу рекурентності обчислення градієнту призводить до майже повного його затухання.

Також, GRU можна вважати узагальненням Simple RNN. Ми можемо отримати Simple RNN блок, поклавши $u_t = 0$ та $r_\tau = 1$ [3].

Отже, GRU архітектура вирішує ті ж проблеми, що і LSTM, але при цьому на кожен часовий проміжок τ ми перераховуємо 6 матриць параметрів замість 8.

1.3 Аналіз генеративних моделей у глибинному навчанні для виявлення прихованих закономірностей у даних

1.3.1 Архітектура encoder-decoder, або автокодувальник (Autoencoder)

Автокодувальником називають глибинну нейронну мережу, мета якої – відтворити ті дані, що подаються їй на вхід. Даний тип мереж складається з двух частин – мережі-кодувальника (encoder) та декодувальника (decoder). Розглянемо послідовно кожну з цих мереж.

Кодувальник – основна частина даної мережі. Мета кодувальника – створити таку функцію

$$g(\cdot): x \to y$$

де $x = \{x_i\},\ y = \{y_j\}$;

$i = \overline{1,n}, j = \overline{1,m}.$

В залежності від величини m кодувальники поділяються на понижуючі (undercomplete) у випадку $m < n$, або підвищуючі (overcomplete) у протилежному випадку. m-вимірний векторний простір до якого належить y, називатимемо прихованим, або латентним простором, а вихідний шар кодувальника – ботлнеком (калька з англ. «bottleneck» – «вузьке місце»).

Декодувальник виконує обернену функцію – тобто відтворює початкові дані з відображення цих даних у латентному просторі, тобто

$$f(\cdot): y \to x$$

Зазвичай декодувальник має ту ж архітектуру, що і кодувальник, але «симетрично» відображену відносно шару, що відповідає за створення латентного простору. Але загалом можна використовувати будь-яку архітектуру.

Як було зазначено вище, дана мережа намагається відтворити те, що їй подається на вхід, але головна ціль при конструюванні такої мережі – дослідити «приховані» закономірності у даних, тому нас не задовольняє варіант того, що мережа просто вивчить вхідні дані і точно відобразить їх на виході. Формально такий випадок можна записати як

$$g\big(f(x)\big) = x$$

Натомість бажано отримати модель, яка повертає лише наближено вхідні дані, користуючись набутими знаннями, відображеними у латентному просторі (отриманими з ботлнека).

Розглянемо детальніше понижуючі автокодувальники. На даний момент це найбільш розповсюджений тип подібних мереж. В даному випадку ключовою задачею є формування «стиснутого» представлення вхідного вектора x шляхом вирішення задачі копіювання вхідних даних. Тобто, наша задача – побудувати правильно латентний простір

$$Y\colon \dim(Y) = m < n = \dim(X)$$

з заданою функцією відображення

$$g(\cdot): X \to Y$$

та функцією реконструкції

$$f(\cdot): Y \to X$$

Для навчання такої мережі введемо наступну функцію:

$$L = L(x, f(y)) = L(x, f(g(x))) \tag{1.30}$$

(1.30) можна інтерпретувати як функцію відстані між x та реконструкцією x та $f\big(g(x)\big)$.

Зокрема, якщо кодувальник лінійний, тобто усі функції активації є лінійними, а функція

$$L = \frac{1}{2}\|x - f(g(x))\|^2$$

то кодувальник буде виконувати декомпозицію x на m головних компонент [4].

У свою чергу, навчати підвищуючий розмірність автокодувальник, дещо складніше, оскільки при навчанні є ймовірність того, що ботлнек буде виконувати роль багатовимірного identity-оператора, а тому мета навчити мережу будь-яким абстракціям, що доповнять картину даних, не буде досягнута.

Заради уникнення таких ситуацій введемо регуляризатор $\mu(h, x)$, де $h = f(x)$ – результат вихідного шару кодувальника. Таким чином, додавши до (1.30) $\mu(h, x)$, отримано наступне рівняння:

$$L_\mu = L(x, f\big(g(x)\big) + \mu(h, x) \tag{1.31}$$

Отже, шляхом введення штрафу за неправильне кодування вхідного вектору, нам не треба обмежувати розмірність латентного простору зради уникнення повного копіювання вхідних даних, тим самим ми змушуємо модель доповнювати вхідний вектор новими властивостями, заради детальної реконструкції вхідних даних.

Таким чином, обравши певні штрафні функції, та додавши у модель нелінійність (наприклад, використовуючи замість «простих» штучних нейронів рекурентні блоки) ми досягнемо високорівневих абстракцій у обробці даних та моделюванні ознак.

Такими функціями похибок оперують розріджені автокодувальники, де штраф є функцією від'ємної логарифмічної правдоподібності, тобто:

$$\mu(h,x) = \mu(h) \ = \sum_{i=1}^{m} \left(\lambda|h_i| - log\frac{\lambda}{2}\right) = -\log_p(h)\,,$$

де $h \sim Laplace(\lambda)$

Виведення формули вище наведено у [4].

Приділимо більше уваги глибині кодувальника та декодувальника. Зазвичай, обидві компоненти автокодувальника роблять одношаровими з нелінійною функцією активації, щоб отримати компактну модель. Але більш глибокі кодувальники та декодувальники насправді мають ряд переваг.

У нейронних мереж прямого розповсюдження, якими і є автокодувальники, є кілька сильних сторін. Одна з таких полягає у тому, що при збільшенні глибини в обмін на зменшення навчальних параметрів моделі надає перевагу у швидкості навчання за рахунок зменшення обчислювальної вартості при представленні деяких функцій та збільшенню рівня абстракції мережі при кодуванні вхідного вектору, що робить останній більш цінним для подальшої обробки.

Також важливою перевагою нетривіальної глибини мережі є її універсальність щодо апроксимації будь-якої функції, про це свідчить універсальна теорема апроксимації [4].

При моделюванні понижуючих автокодувальників, експериментально підтвердилося те, що архітектури, що базуються на глибинних нейронних мережах досягають набагато більшого показника стиснення вхідних даних, ніж лінійні кодувальники та одношарові [4].

1.3.2 Ймовірнісне породжуюче моделювання. Варіаційні автокодувальники

У попередньому пункті ми розглядали звичайні (детерміновані) автокодувальники, функціонал яких є дещо обмеженим в силу присвоєння кожному екземпляру вибірки даних сталому коду.

Інакше кажучи, автокодувальники – це звичайні мережі прямого розповсюдження, тобто для побудови таких мереж можна використовувати ті ж методи та функції похибок, що й при створенні «звичайних» мереж, на кшталт тих, що були розглянуті у п. 1.2. Тобто, можемо використати від'ємну логарифмічну правдоподібність, визначену формулою (1.8) та підставивши її у (1.30) або (1.31), якщо використовуємо підвищуючі кодувальники та регуляризацію.

Оскільки для автокодувальника вхід та ціль – один та той же вектор x, то «прив'язавши» функцію похибок до виходу останнього шару мережі, можна стверджувати, що декодувальник при подачі на вхід коду z повертає умовний розподіл $p(x/z)$, а функція похибок визначає правдоподібність такої «прив'язки».

Наприклад, у мережах прямого розповсюдження, якщо x – вектор з $\mathbb{R}^n$, то застосовуючи середньоквадратичну похибку (при умові лінійних функцій активації) умовний розподіл $p(x/z)$ буде являти собою нормальний розподіл величини $f(h) \sim \mathcal{N}(x, MSE(f(h), x))$.

Більш корисно було би піти далі у цьому напрямку та навчити мережу відтворювати закон розподілу навчальних даних для синтетичної генерації вибірок з генерального розподілу. Тобто ми відходимо від поняття вектор-коду до розподілу коду.

Заради таких цілей були створені варіаційні автокодувальники, котрі оперують не детермінованими закодованими значеннями вхідних даних x, а ймовірнісним розподілом $q(z/x)$, де z – випадковий вектор у латентному просторі, а декодувальник оперує умовною ймовірністю $p(x/z)$.

В таких мережах, оскільки ми оперуємо умовними ймовірнісними розподілами, то необхідно використовувати фреймворк варіаційних наближень. Варіаційні наближення ґрунтуються на теоремі Байєса, Розберемо це поняття на прикладі.

Нехай набір даних x був породжений мережею з вектором параметрів θ та вхідним набором даних z, що в свою чергу був породжений з деякого апріорного розподілу $p(z/\theta)$. Тобто, маючи вектор параметрів θ та відомий апріорний розподіл $p(z/\theta)$ породжуємо прихований вектор z, а після цього з умовного розподілу $p(x/z,\theta)$ породжуємо x.

Обидва розподіли ми знаємо, але щоб отримати апріорний розподіл $p(x/\theta)$, необхідно порахувати наступний інтеграл:

$$p(x/\theta) = \int_Z p(x/z,\theta)\, p(z/\theta) dz$$

Але інтегрувати по всім можливим значенням Z – складна задача. Заради вирішення таких проблем можна використати EM-підхід, що базується на використанні теореми Байєса:

$$p(z/x\,,\theta) = \frac{p(x/z,\theta)p(z/\theta)}{p(x/\theta)}$$

EM-підхід полягає у тому, що ми спочатку беремо оцінку математичного сподівання по z, а потім з таким зафіксованим z перераховуємо параметр θ, взявши максимальне значення.

Але якщо розподіл $p(z/x,\theta)$ достатньо складний, то ми не зможемо використати алгоритм вище, оскільки основною проблемою буде обрахунок математичного сподівання. Такий випадок – не рідкість, оскільки навіть найпростіша двошарова нейронна мережа з нелінійністю типу сигмоїди вже оперує складними нетривіальними ймовірнісними розподілами [3].

Щоб обійти таку проблему використаємо один з основних принципів математичного сподівання – складне апроксимуємо простим. В даному випадку замість відновлення складного розподілу ми оперуємо класом простих розподілів і серед сімейства розподілів та їх параметрів знаходимо такі, що отриманий розподіл буде максимально схожий за поведінкою до досліджуваного. Це і є сутність концепції варіаційного наближення.

Для таких цілей необхідно також визначитись з функцією втрат. Оскільки ми оперуємо випадковими факторами, то перейдемо до математичного сподівання втрат відносно z:

$$\mathbb{E}_{z\sim q(z/x)}L(z,q) = \mathbb{E}_{z\sim q(z/x)}\log p(x,z) - D_{KL}(q(z/x) \parallel p(z)) \qquad (1.32)$$

У даній формулі $p(x,z)$ – сумісний розподіл x та z, $D_{KL}(P \parallel Q)$ – дивергенція Куллбака-Лейблера:

$$D_{KL}(P \parallel Q) = \int_{-\infty}^{\infty} p(x)\log\left(\frac{p(x)}{q(x)}\right)dx \qquad (1.33)$$

(1.33) визначає ступінь «несхожості» розподілів P та Q. Розглянемо детальніше чому це дійсно так.

Повернемося до алгоритму EM. Нехай x – фактичні значення досліджуваного ряду, z – значення прихованого вектору. Нагадаємо, що EM-

алгоритм припускає, що ми можемо обрахувати розподіл $p(x, z \,/\, \theta) = p(z/x\,, \theta)p(x/\theta)$, причому максимізуємо $p(x/\theta)$ відносно параметрів θ.

Візьмемо логарифм від обох частин рівності вище та виразимо $\log p(x/\theta)$:

$$\log p(x/\theta) = \log p(x, z/\theta) - \log p(z/x, \theta)$$

Тепер візьмемо математичне сподівання відносно z:

$$\int_Z q(z) \log p(x/\theta)\, dz = \int_Z q(z) \log p(x, z/\theta)\, dz - \int_Z q(z) \log p(z/x, \theta)\, dz$$

Інтеграл зліва рівний $\log p(x/\theta)$, оскільки по відношенню до z – це константа, а до правої частини додаємо та віднімаємо $\int_Z q(z) \log q(z) dz$:

$$\log p(x/\theta) = \int_Z q(z)\big(\log p(x, z/\theta) - \log q(z) - \log p(z/x, \theta) + \log q(z)\big)dz$$

$$= \int_Z q(z) \log\left(\frac{p(x, z/\theta)}{q(z)}\right) dz - \int_z q(z) \log\left(\frac{p(z/x, \theta)}{q(z)}\right) dz$$

Користуємось тим, що $p(x, z/\theta) = p(x/z, \theta)p(z/\theta)$:

$$\log p(x/\theta) = \int_Z q(z) \log\left(\frac{p(x/z, \theta)p(z/\theta)}{q(z)}\right) dz - \int_z q(z) \log\left(\frac{p(z/x, \theta)}{q(z)}\right) dz$$

Як бачимо, останній доданок справа – це дивергенція Куллбака-Лейблера. Цей показник завжди невід’ємний. Тому вираз

$$\int_Z q(z) \log\left(\frac{p(x/z,\theta)p(z/\theta)}{q(z)}\right) dz$$

є нижньою оцінкою величини $\log p(x/\theta)$ [3]. Таку оцінку називають нижньою варіаційною оцінкою (evidence lower bound).

Позначимо

$$\int_Z q(z) \log\left(\frac{p(x/z,\theta)p(z/\theta)}{q(z)}\right) dz = ELBO(q,\theta)$$

тоді:

$$ELBO(q,\theta) = \log p(x/\theta) - D_{KL}(q(z) \parallel p(z/x)) \tag{1.34}$$

В цілому, максимізуючи (1.34) по параметру θ ми «наближаємо» $q(z)$ до $p(z/x)$.

Покажемо, що (1.32) тотожна до (1.34). Користуючись формулами умовної ймовірності, перепишемо (1.32) у вигляді:

$$\mathcal{L}(q) = \mathbb{E}_{z\sim q(z/x)} \log p(x/z) - D_{KL}(q(z/x) \parallel p(z)) \tag{1.35}$$

Оскільки $D_{KL}(q(z/x) \parallel p(z))$ є близькою до нуля при однакових розподілах $q(z/x)$ та $p(z)$, а також виконується

$$\log p(x/z) \leq \log p(x)$$

то відповідно верхня межа функції втрат (в даному випадку краще називати $\mathcal{L}(q)$ функцією виграшу, оскільки ми прагнемо її максимізувати щоб

відтворити розподіл x) буде також $\log p(x)$ [4]. Тобто ми прийшли до одного і того ж результату.

Для проведення навчання необхідно, щоб усі функції активації були диференційовними відносно параметрів моделі оскільки ми будемо навчати модель породжувати параметри розподілу методом Rprop відносно вибірки $z{\sim}q(z/x) = q(z; f(x;\theta))$, щоб отримати градієнт відносно θ. При цьому наближені обчислення математичного сподівання $\log p(x/z)$ та дивергенції Куллбака-Лейблера при фіксованому θ можна виконати методом Монте-Карло.

1.4 Проблематика предметної області. Постановка задачі дослідження

У попередніх пунктах були розглянуті основні методи аналізу та дослідження часових рядів. «Класичні» методи, що були розглянуті у п .1.1 оперують стаціонарними рядами, тому будь-який часовий ряд перед моделюванням необхідно приводити до такого шляхом чисельного диференціювання (застосовуючи кінцеві різниці), щоб позбавити ряд тренду, відмінного від константи; масштабування, логарифмування, тощо. Також необхідно вручну обирати ступінь авторегресії та ковзного середнього, аналізуючи показники автокореляції. Тобто втручання людини у процес інжинірингу даних займає більшість часу у процесі моделювання та визначатиме адекватність моделі в цілому.

У свою чергу, методи, що використовують машинне та глибинне навчання в основному потребують лише масштабування даних (в залежності від функції активації) та перетворення самої форми подачі ряду від вигляду «час»-«значення» до «значення за попередні періоди»-«значення за поточний період», тобто подати у вигляді «ознаки-результуюче значення».

Основною проблемою такого представлення є інваріантність щодо зафіксованих значень ряду відносно часу. Простіше кажучи, таке

представлення ряду припускає, що кожне значення ряду фіксується з однаковим інтервалом, хоча це не завжди так.

Отже, наша основна задача – розробка моделі часового ряду, що була би чутлива до неоднакових інтервалів фіксації значень та могла би знаходити приховані закономірності у даних. При цьому бажано мінімізувати втручання людини у процес обробки даних, залишивши з вимог лише приведення до одиничного інтервалу досліджувані значення.

1.5 Висновки

У даному розділі ми дослідили предметну область нашої задачі та переглянули низку широковідомих алгоритмів та моделей аналізу часових рядів.

У п. 1.1. були розглянуті «класичні» моделі, що слугують для обробки часових рядів, а саме:

а) ARMA-модель, що використовує для обрахування поточних значень лінійну комбінацію попередніх значень досліджуваного ряду та показників ковзного середнього. Для адекватності моделі необхідно використовувати її для прогнозування стаціонарних процесів.

б) ARIMA та ARCH/GARCH – моделі. Дані моделі використовуються для нестаціонарних процесів. Перші позбавляються від нестаціонарності шляхом дискретного диференціювання (тобто за допомогою оператора кінцевих різниць приводять ряд до стаціонарного), другі моделі використовують «ковзну дисперсію» для моделювання гетероскедастичності. Такі моделі широко використовуються для аналізу фінансових часових рядів, особливо тих показників, де потрібно досліджувати волатильність цінних паперів.

У п. 1.2 дослідили методи машинного навчання для обробки та аналізу послідовностей. Оскільки глибинні нейронні мережі можуть моделювати нелінійні залежності у даних, то для обробки послідовностей було вирішено

розглянути один їхній клас – рекурентні нейронні мережі (РНМ), та оглянули основні їх різновиди:

а) Звичайні рекурентні мережі продовжують рухатися у тому ж напрямку, що й класичні моделі, але їх глибина та нелінійність між дозволяє підвищити рівень абстракції і маніпулювати не лише лінійною залежністю від попередніх значень ряду, а й більш складними типами залежностей. Нажаль, у Simple RNN є низка недоліків – ці мережі достатньо повільно навчаються оскільки параметри треба оптимізувати не лише від ітерації, до ітерації, а й відносно часу, що значно зменшує швидкість навчання через збільшення кількості обчислень. Інший недолік – проблема затухаючого градієнта, або напроти – вибуху градієнту, що виникає при рекурсивному виклику градієнтів та пов’язана з особливостями ініціалізації вектору параметрів.

б) Мережі з довгою короткостроковою пам’яттю (LSTM). LSTM мережі збільшують глибину рекурсії, додаючи до кожного блоку управляючі модулі, що називають «гейтами». Дані модулі дозволяють обирати мережі, які часові лаги потрібно «запам’ятовувати», що значно збільшує обсяг пам’яті. Також в силу своїх можливостей, архітектура здатна вирішити проблему затухаючих градієнтів, але їхня складна конструкція робить дані мережі дуже дорогими з точки зору обчислень.

в) Рекурентні вентильні мережі (GRU), що слугують компромісом між відносною швидкодією простих рекурентних мереж та LSTM, спрощуючи архітектуру останньої. На даний момент це найновіша технологія (не вважаючи технологіями нові модифікації LSTM, оскільки принцип їхньої роботи залишається незмінним) рекурентних мереж, що широко використовується та витісняє LSTM, що стали стандартом при побудові моделей типу послідовність-послідовність.

Також у п. 1.3 ми розглянули такі нові архітектури нейронних мереж типу кодувальник-декодувальник, як:

а) Звичайні автокодувальники. Це мережі, що перетворюють вхідний вектор сам в себе через проходження крізь низку нелінійностей у шарах даної нейромережі. Але головна мета, з якою створюються автокодувальники – це отримання проміжного стану, що називається кодом і являє собою нелінійне перетворення вхідного сигналу, що зберігає найбільше інформації про досліджуваний процес, але не копіює його. Дані мережі створюються заради пошуку прихованих закономірностей у даних, що необхідно обробити. Також, ми розглянули класифікацію кодувальників за розмірністю прихованого стану: понижуючі та підвищуючі.

б) Варіаційні автокодувальники, що відрізняються від зазначених вище тим, що моделюють не конкретний вектор-код, а моделюють цей вектор як випадковий з заданого сімейства розподіл. Також вхідний вектор інтерпретується як випадковий з іншого сімейства розподілів. Для варіаційних автокодувальників використовуються нестандартні функції похибок, що базуються на результатах варіаційного виведення та наближення.

2 МОДЕЛЮВАННЯ ДИНАМІКИ ЧАСОВИХ РЯДІВ МЕТОДАМИ НЕЙРОННИХ МЕРЕЖ

2.1 Основні терміни теорії диференціальних рівнянь. Метод спряжених ЗДР для вирішення задач отпимізації

Оскільки ми займатимемося задачами відновлення динаміки, то для моделювання динаміки нам необхідно використовувати диференціальні рівняння та системи цих рівнянь. У даній роботі використовуватимемо лінійні звичайні диференціальні рівняння. Наведемо означення.

Звичайним диференціальним рівнянням першого порядку називається рівняння, що містить похідну невідомої функції.

Порядком рівняння називається максимальний порядок похідної, що в це рівняння входить [6].

У даній роботі ми будемо використовувати рівняння першого порядку, тобто наступного вигляду:

$$\frac{dy}{dx} = f(x, y) \tag{2.1}$$

Розв’язок такого рівняння:

$$F(x) + C : \frac{dF(x)}{dx} \equiv f(x, y) \tag{2.2}$$

Для того, щоб з сімейства функцій, зазначеного в (2.2) виокремити конкретне рівняння, необхідно розв’язати систему вигляду:

$$\begin{cases} \frac{dy}{dx} = f(x, y), \\ y(x)|_{x=x_0} = y_0 \end{cases} \tag{2.3}$$

Систему (2.3) називають задачею Коші.

Введемо також поняття спряженого звичайного диференціального рівняння, запропоноване математиком Понтрягіним у [7]:

Спряженим до рівняння (2.1) назвемо рівняння вигляду:

$$\frac{da}{dx} = -a(x)\frac{df(y(x),x)}{dy(x)} \tag{2.4}$$

Рівняння (2.4) є основою методу спряжених рівнянь (у зарубіжній літературі метод має назву adjoint sensitivity method) для оптимального диференціювання при вирішенні задач чисельної оптимізації. При цьому зв'язок між (2.1) та (2.4) полягає у наступному:

Нехай маємо (2.1) – лінійне та задане у явному вигляді, позначене як

$$h(y, y', x, p) = h = y' - A(p)y - b(p) = 0$$

тоді:

$$\frac{da}{dx} = -a(x)\frac{df(y(x),x)}{dy(x)}, \tag{2.4}$$

$$y' = A(p)y + b(p), \tag{2.5}$$

$$a' = A^*(p)a - {f_x}^* \tag{2.6}$$

де через символ «*» позначений оператор спряження, p – деякий параметр [8]. В загальному випадку даний метод використовується для задач типу:

$$\min_p F(x,p) = \int_0^T f(x,p,t)dt \tag{2.7}$$

за обмежень:

$$h(x, \dot{x}, p, t) = 0, \tag{2.8}$$

$$g(x(0), p) = 0 \tag{2.9}$$

Бачимо, що умова (2.9) задає початкову точку, тобто ми маємо оптимізувати $F(x, p)$, віднайшовши x через розв'язок задачі Коші.

2.2 Перехід від дискретного до неперервного часу у рекурентних нейронних мережах

Як було зазначено вище, рекурентні нейронні мережі оперують дискретним часом та рівними часовими проміжками. Тому, у загальному випадку можна записати, що перетворення прихованого стану має наступний вигляд:

$$h_{t+1} = h_t + f(h_t, \theta) \tag{2.10}$$

Тут і надалі $t \in \overline{0, T}$.

Нехай відбудеться перехід до менших часових проміжків та додавати взамін більше шарів, тоді (2.10) набуває наступного вигляду:

$$\lim_{\Delta t \to 0} \frac{h_{t+\Delta t} - h_t}{\Delta t} = \frac{dh}{dt} = f(h, t, \theta) \tag{2.11}$$

Таким чином, у граничному випадку з нескінченно кількістю рекурентних шарів та нескінченно малим часовим кроком було отримано «аналогову» нейронну мережу, що моделює звичайне диференціальне рівняння, аналогічне (2.1). Поклавши $h(t) = h_t \ \forall \ t$ на кожному з дискретних часових значень було виведено систему рівнянь, аналогічну (2.3).

Використання ЗДР для побудови таких моделей (домовимось називати їх ЗДР-мережами) має ряд переваг. По-перше, мережа стає більш чутливою до часу, що відповідає моментам фіксації значень ряду, тобто можна використовувати для моделювання часових рядів, значення яких фіксувались з різною інтенсивністю впродовж спостереження.

По-друге, використання диференціальних рівнянь дозволяє нам, очевидно, розв'язувати задачу оптимізації параметрів моделі, як задачу (2.7)-(2.9). Метод спряжених рівнянь, анонсований у попередньому пункті може виявитись значно швидшим, ніж зворотне розповсюдження похибки, оскільки не потребує зберігання параметрів моделі при прямому розповсюдженні похибки [7].

2.3 Навчання ЗДР-мереж методом спряжених рівнянь.

Найбільшою складністю, що виникає при навчанні глибинних нейронних мереж, є велика кількість параметрів моделі, по кожному з яких необхідно оптимізувати функцію втрат (похибок). Градієнтні методи, такі як Rprop, як бачимо з формул п. 1.2.1., мають запам'ятовувати значення градієнтів впродовж двох послідовних ітерацій для проведення оптимізації результуючої функції похибок.

Очевидно, що при збільшенні шарів та зменшенні часового кроку, нашу ЗДР-мережу буде складно навчити з точки зору використання оперативної пам'яті, через її громіздкість. Тому, необхідно використовувати методи, що є більш ефективні з точки зору використання пам'яті та були такими ж ефективними.

Нехай маємо функцію, що залежить від неперервного часу t, прихованого стану моделі z, та виражає похибку при прогнозуванні значення часового ряду у момент $t = t_1$, знаючи значення ряду у момент t_0. Така функція після виконання відповідних підстановок та інтегрування (2.11) набуває наступного вигляду:

$$L(z, t = t_1) = L\big(z(t_1)\big) = L\left(z(t_0) + \int_{t_0}^{t_1} f(z, t, \theta)dt\right) \tag{2.12}$$

У функції втрат одним з параметрів фігурує прихований стан z, що очевидно, залежить від вектору параметрів моделі θ. Тому для оптимізації значень θ необхідно знати залежність z від θ, для обчислення градієнту функції втрат по θ.

Застосуємо підхід спряжених рівнянь і визначимо $\frac{dL}{dz} = a(t)$ як спряжене рівняння. Перепишемо (2.4) у наступному вигляді:

$$\frac{da}{dt} = -a(t)\frac{df(z(t), t, \theta)}{dz(t)} \tag{2.13}$$

Таким чином, задача оптимізації (2.12) може бути вирішена, розв'язавши рівняння (2.13) відносно t, щоб отримати рівняння траєкторії $z(t)$. Але, знаючи $z(t_1)$ ми можемо перерахувати $z(t)$ неявно через $a(t)$ зворотно часу і одразу отримати градієнт функції похибок по θ,зробивши ряд припущень.

Нехай

$$\frac{d\theta}{dt} = 0$$

тобто θ не залежить від значень часу. Таке припущення працює у рекурентних мережах, оскільки параметри моделі є інваріантними від часу і спільними для всіх часових станів прихованих шарів мережі.

Далі порахуємо градієнти параметрів функції похибок відносно часу:

$$\frac{d}{dt}\begin{bmatrix} z \\ \theta \\ t \end{bmatrix} = \begin{bmatrix} f(z,\theta,t) \\ 0 \\ 1 \end{bmatrix} = f_{aug}(z,\theta,t) \tag{2.14}$$

Визначимо тепер спряжені рівняння для (2.14):

$$a_{aug} = \begin{bmatrix} a \\ a_\theta \\ a_t \end{bmatrix} = \begin{bmatrix} \frac{\partial L}{\partial z} \\ \frac{\partial L}{\partial \theta} \\ \frac{\partial L}{\partial t} \end{bmatrix} \tag{2.15}$$

Для застосування «ланцюгового правила» диференціювання складних функцій (chain rule), необхідно порахувати якобіан $f_{aug}(z,\theta,t)$ відносно всіх параметрів:

$$\frac{df_{aug}(z,\theta,t)}{d[z \quad \theta \quad t]} = \begin{bmatrix} \frac{\partial f}{\partial z} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{2.16}$$

Нарешті, підставимо все необхідне в (2.4) та отримаємо:

$$\frac{da_{aug}}{dt} = -{a_{aug}}^T \frac{df_{aug}(z,\theta,t)}{d[z \quad \theta \quad t]} = -\begin{bmatrix} a\frac{\partial f}{\partial t} & a\frac{\partial f}{\partial \theta} & a\frac{\partial f}{\partial t} \end{bmatrix} \tag{2.17}$$

Покладемо

$$\left.\frac{\partial L}{\partial \theta}\right|_{t=T} = a_\theta(T) = 0$$

Отже, з формули (2.17) бачимо наскільки даний метод ефективний, оскільки потребує введення лише одного додаткового рівняння і не потребує явних обрахунків градієнтів функції похибок по параметрам моделі.

Таким чином, маємо:

$$\frac{dL}{d\theta} = -\int_{T}^{t_0} a(t)\frac{\partial f(z(t),t,\theta)}{\partial \theta}\,dt, \tag{2.18}$$

$$\left.\frac{dL}{dt}\right|_{t=T} = -a(T)f(z(T),T,\theta), \tag{2.19}$$

$$\left.\frac{dL}{dt}\right|_{t=t_0} = a_t(T) - \int_{T}^{t_0} a(t)\frac{\partial f(z(t),t,\theta)}{\partial t}\,dt \tag{2.20}$$

Отже, маючи пораховані градієнти функції похибок, можемо провести оптимізацію будь-яким методом оптимізації, наприклад, методом зворотного поширення похибки, не перераховуючи значення градієнтів для кожного прихованого шару у конкретний момент часу [9].

Зобразимо графічно процес оптимізації та відшукання градієнтів цільової функції:

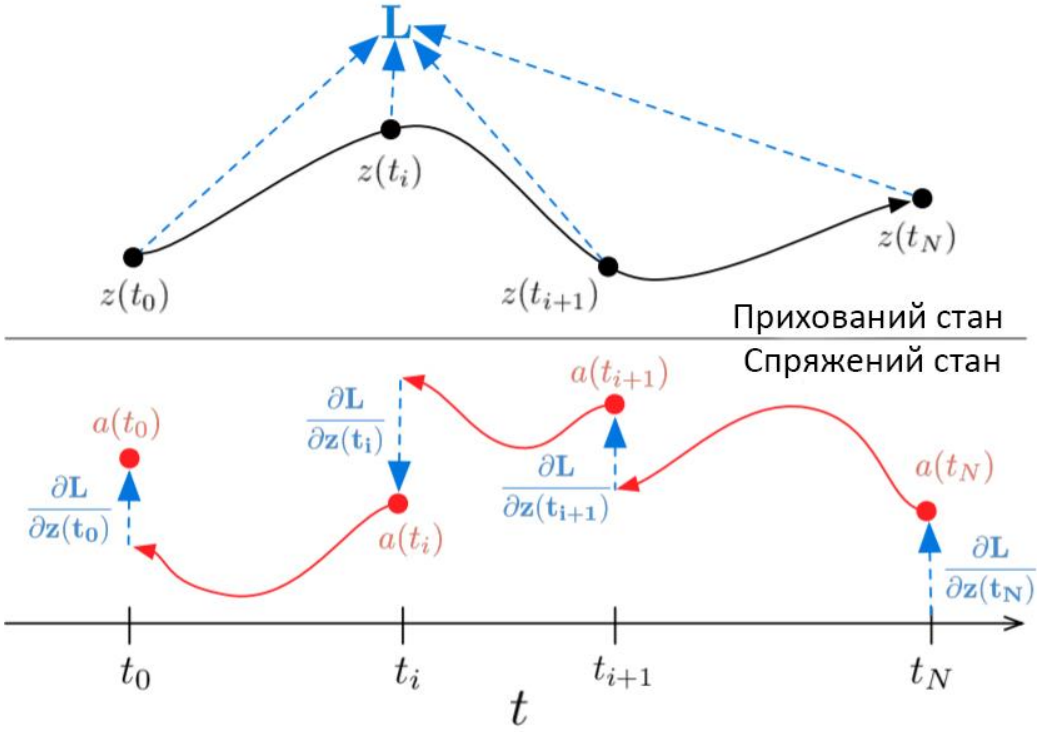


Рисунок 2.3 – Графічна інтерпретація методу спряжених рівнянь

Як бачимо з рисунку, при кожному розрахунку градієнту функції похибки ми коригуємо значення параметрів через коригування спряжених рівнянь відносно спостережуваних точок.

2.4 Генеративне моделювання часових рядів методами ЗДР.

Як обговорювалося раніше, більшість часових рядів надходять до нас у вигляді значень, взятих не з однаковими часовими інтервалами. Через це, на відміну до «авторегресивного» підходу (до таких моделей можна навести й рекурентні нейронні мережі, оскільки принцип їх роботи полягає у відшуканні залежності поточного значення ряду від спостережуваних історичних значень), нам потрібна модель, яка була би чутливою і до нерівномірності спостережуваних значень відносно загального інтервалу спостережень.

Для таких цілей ми можемо використати генеративну ЗДР-модель, інтегрувавши відновлення динаміки ряду методом ЗДР у ймовірнісну породжуючу модель, описану в п. 1.3.2.

Нехай, для простоти запису, розв'язок ЗДР , що фігурує у (2.12), позначимо як $Sol(z, f, \theta, t)$. Тепер маємо наступну генеративну модель:

$$z_{t_0} \sim p(z_{t_0}), \tag{2.21}$$

$$\forall i = \overline{1,n}\ z_{t_i} = Sol(z_{t_0}, f, \theta_f, t_i), \tag{2.22}$$

$$\forall i\ x_i \sim p(x_i / z_{t_i}, \theta_x) \tag{2.23}$$

Припускаємо, що функція f не залежить від часу та діє наступним чином:

$$f(z(t), \theta_f)\big|_{t=\tau} = \frac{\partial z(t)}{\partial t}(\tau) \tag{2.24}$$

Функцію (2.24) будемо параметризувати ЗДУ-мережею. Оскільки f інваріантна від часу, то кожна прихована траєкторія, що моделюється (2.21)-(2.23) буде унікальною [8].

Оскільки мережа, яку ми конструюємо є породжуючою, то будемо використовувати фреймворк варіаційних автоенкодерів.

Для побудови моделі необхідно визначитись з архітектурою кодувальника та декодувальника. Оскільки ми моделюємо приховані траєкторії спираючись не лише на розподіл першої точки траєкторії, а й на проміжні точки у момент часу $t = \tau$, тобто це всі точки у інтервалі $(t_1;\ \tau)$, то для кодувальника можна використовувати рекурентні мережі, але з певними модифікаціями, що роблять їх чутливими до нерівномірності спостережень у часі.

2.4.1 Приховані ЗДР-мережі (Latent ODE). Гібридні LODE-GRU мережі.

У даній секції ми представимо модифікації рекурентних мереж типу GRU, що використовують апарат ЗДУ для прогнозування значень прихованих траєкторій. Така архітектура дозволяє генерувати значення у проміжних точках між спостереженнями. Тобто, за допомогою такої методики ми генеруємо нові значення ряду там, де часовий проміжок між двома сусідніми спостереженнями великий.

Таким чином, ми вирішили проблему інваріантності параметрів GRU відносно часу спостережень, доповнивши «справжні» точки проміжними значеннями з моделі динаміки.

Тобто, рівняння такого блоку отримується шляхом модифікації системи (1.26)-(1.29) та має наступний вигляд:

$$h''_\tau = Sol(f_\theta, h_{\tau-1}, t_{\tau-1}, t_\tau), \tag{2.25}$$

$$u_\tau = \sigma(W_{xu}x_\tau + W_{hu}h''_\tau + b_u), \tag{2.26}$$

$$r_\tau = \sigma(W_{xr}x_\tau + W_{hr}h''_\tau + b_r), \tag{2.27}$$

$$h'_\tau = \tanh\big(W_{xh}x_\tau + W_{hh'}(r_\tau \odot h''_\tau)\big), \tag{2.28}$$

$$h_\tau = u_t \odot h''_\tau + (1 - u_\tau) \odot h'_\tau \tag{2.29}$$

Блок, що описується (2.25) генерує набір точок відповідно траєкторії досліджуваного процесу за умов перетворень процесу рівняннями (2.26)-(2.29), що відображають траєкторію на інший простір, який назвемо латентним (прихованим). Тому (2.25) називається Прихованим ЗДУ (Latent ODE), а сам тип архітектури типу (2.25)-(2.29) – ODE-RNN [9].

Модифікуємо дану мережу, щоб отримати ймовірнісні розподіли прихованих траєкторій. Візьмемо один з найпростіших випадків – нехай вихідний вектор кодувальника буде багатовимірним гаусівським вектором з математичним сподіванням у період останнього спостереження τ рівним μ_{z_0} та дисперсією σ_{z_0}.

Тому до системи (2.25)-(2.29) додамо шар, що описується формулою:

$$z_0 = g(h_\tau, \theta) \sim \mathcal{N}(\mu_{z_0}, \sigma_{z_0}) \quad (2.30)$$

Таким чином, формули (2.25)-(2.30) описують варіаційний кодувальник.

2.4.2 Генеративне моделювання пуасонівських процесів за допомогою LODE.

У попередньому пункті ми розглянули найбільш розповсюджений метод отримання прихованого вектору – моделювання його як гаусівського випадкового вектору. Для відтворення початкового ряду з прихованого стану можна застосувати формули (2.21)-(2.23) і отримати базовий автокодувальник. Зобразимо графічно як він працює (рис 2.4):

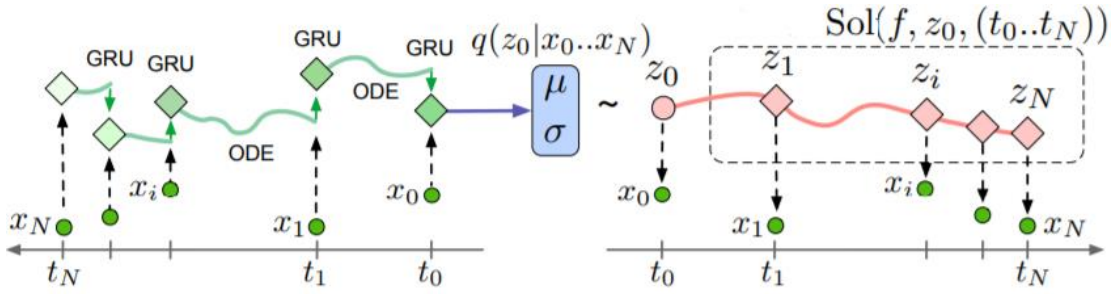


Рисунок 2.4 – Графічна інтерпретація роботи варіаційного автокодувальника

Але, іноді часові проміжки, що відповідають моментам фіксування спостережень, можуть також містити корисну інформацію про часовий ряд. Тобто, знаючи частоту фіксації події упродовж періоду спостережень можна домогтися більшої відповідності моделі досліджуваному процесу.

Наприклад, якщо у нас є часовий ряд, що відповідає даті звернення хворих до реєстратури поліклініки. Якщо ми звернемо увагу на те, що починаючи з якогось часу $t = t_0$ кількість звернень людей з однаковими симптомами стрімко збільшується, а час між сусідніми зафіксованими значеннями зменшується, то така інформація допоможе нам при створені моделі розповсюдження хвороби на території, що оточує цю поліклініку.

Для створення таких моделей застосовують неоднорідні пуасонівські процеси, тобто такі, що інтенсивність появи події (параметр λ розподілу) є функцією від часу спостережень t[10].

Для таких цілей ми можемо додати до декодувальника ще одну LODE-мережу, що буде параметризувати параметр $\lambda(t)$.

Для цих цілей, доповнимо вектор z вектором z_λ та інтегралом $\lambda(t)$ відносно часу. Тоді також $\lambda(t) = \lambda = \xi(z_\lambda)$. Далі наведемо похідну по «доповненому» вектору z:

$$\frac{d}{dt}\begin{bmatrix} z \\ z_\lambda \\ \int_0^t \lambda(\tau)d\tau \end{bmatrix} = \begin{bmatrix} f \\ f_\lambda \\ \lambda(t) \end{bmatrix}$$

Тоді декодувальник можна описати наступною системою:

$$z_{t_0} \sim p\left(z_{t_0}\right), \tag{2.31}$$

$$\lambda(t) = Sol\left(z_{t_0}, f_\lambda, \theta_{f_\lambda}, t\right), \tag{2.32}$$

$$\forall i = \overline{1,n}\ t_i \sim PoissProcess\left(\lambda(t)\right), \tag{2.33}$$

$$\forall i = \overline{1,n}\ z_{t_i} = Sol\left(z_{t_0}, f_z, \theta_{f_z}, t_i\right), \tag{2.34}$$

$$\forall i\ x_i \sim p(x_i / z_{t_i}, \theta_x) \tag{2.35}$$

2.4.3 Вибір функції похибок.

Отже, після вибору архітектури мережі необхідно обрати функцію, яку треба оптимізувати, тобто функцію похибок.

Ми користуємося фреймворком варіаційних автокодувальників, тому будемо модифікувати функцію похибок (1.34).

Для цього визначимо функцію логарифмічної правдоподібності при відтворенні x з прихованого вектору z. Оскільки до загальної мережі було також вирішено додати мережу, що моделює параметри пуасонівського

процесу для виявлення природи розподілу часових міток в даних, то додамо у функцію втрат також залежність від часових міток.

Оскільки параметр λ залежить від z та t, то логарифмічна правдоподібність для відшукання λ:

$$\log(t_1, \dots t_N/t_{min}, t_{max}, \lambda) = \sum_{i=1}^{N} \log \lambda(t_i) - \int_{t_{min}}^{t_{max}} \lambda(t)dt \qquad (2.36)$$

Також, відтворення розподілу часових проміжків дозволяє замість максимізації умовної маргінальної правдоподібності $p(x_1, \dots, x_N/t_1, \dots, t_N, \theta)$, використати сумісну маргінальну правдоподібність $p(x_1, t_1, \dots, x_N, t_N, \theta)$ [10]. Таким чином, отримуємо функцію похибок виду:

$$\mathcal{L}(\theta, \phi) = \mathbb{E}_{z_0 \sim q_\phi(z_0/\{x,t\})} \log p_\theta(x) - D_{KL}(q_\phi(z_0/\{x,t\}) \parallel p_\theta(z_0))$$

Позначимо це рівняння через (2.37). Навчання моделі будемо проводити методом зворотного поширення похибки, а в LODE-модулях – методом спряжених рівнянь.

2.5 Висновки

В даному розділі була розглянута концепція відновлення динаміки часових рядів використовуючи нейронні мережі. Заради цієї цілі було вирішено використати гібридну ймовірнісну модель-автокодувальник, використовуючи для закодовування моделі рекурентні нейронні мережі. Але, оскільки параметри мережі не залежать від часу, а сама мережа оперує лише історичними даними, отриманими дискретизацією спостережуваного процесу відносно часу, то було вирішено модифікувати рекурентну нейронну мережу, додавши до неї блок, що моделює динаміку прихованого стану за допомогою ЗДР.

Далі, оскільки додаткова задача полягає досліджені у природи розподілення часових міток, що фігурують в історичних даних, то для таких цілей була введена додаткова нейронна мережа до складу моделі, що методами ЗДР відновлює залежність інтенсивності появи зафіксованих вимірів у часі.

Таким чином, було вирішено побудувати мережу-автокодувальник, де кодувальник описується системою (2.25)-(2.30), а декодувальник – (2.31)-(2.35). При цьому функція витрат задається (2.37), навчання мережі проведемо алгоритмом зворотного розповсюдження похибки, градієнти в ЗДР-шарах мережі для збільшення ефективності використання пам’яті, обчислюються методом спряжених рівнянь.

3 АРХІТЕКТУРА ТА АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ

3.1 Вибір платформи та мови реалізації

Одним із найважливіших кроків було обрання мови програмування, на якій буде зручно реалізувати всі теоретичні викладки п.1 та п.2. Критеріями вибору мови програмування та платформи реалізації проекту.

Було вирішено використовувати мову Python 3 та фреймворк тензорних обчислень Tensorflow 2. Вибір на користь даної мови і даної бібліотеки пояснюється наступними чинниками.

Python – одна з найпопулярніших мов програмування, що застосовується у сфері машинного навчання. Такого статусу ця мова здобула завдяки своєму простому синтаксису, що наближає її до мови «псевдокоду», що у свою чергу дозволяє концентруватися більше на «ідейній» складовій проекту, ніж на «тонкощах» реалізації програмних модулів. Але до слабких сторін відносять відносну повільність роботи програм (у порівнянні з мовами C та C++). Повільність пояснюється самою природою даної мови. Python традиційно відносять до мов, що інтерпретуються, або до мов-сценаріїв, тобто для того, щоб програма написана цією мовою виконалась, у системі необхідна наявність проміжного «шару» - інтерпретатора. Саме такий метод трансляції «на лету» у машинний код і є основною причиною того, що Python-код виконується у 10-100 повільніше, ніж код, що виконує аналогічні дії, написаний на C [11].

Після того, як обрано мову програмування, необхідно визначитись з вибором бібліотеки, або фреймворка, що дозволив би спростити процес написання архітектур моделей, зазначених у розділах 1 та 2. Для таких цілей використовуються фреймворки тензорних обчислень Tensorflow, PyTorch та MxNet. Серед цих трьох бібліотек було вирішено зупинитися на Tensorflow, оскільки модулі, що використовують дану бібліотеку, використовували найменшу кількість пам’яті графічного та центрального процесорів, ніж модулі, написані на тих бібліотеках, що залишилися у списку [12]. Обговоримо детальніше даний фреймворк.

Tensorflow – бібліотека тензорних обчислень, що широко використовується для імплементації методів машинного та глибинного навчання завдяки вбудованому апарату символьної математики. Даний апарат дозволяє, наприклад, не обчислювати кожного разу числові похідні при тренуванні ШНМ методами оберненого розповсюдження похибки, а диференціювати «напряму» функції активацій у її шарів. Також, починаючи з версії 2.0 у даний фреймворк інтегрована бібліотека Keras у якості високорівневого API. Зазначимо також, що back-end Tensorflow здебільшого написаний на C++ та використовує можливості платформи Nvidia CUDA та CuDNN, що дозволяє вкупі з представленням операцій у вигляді графу обчислень, виконувати операції над тензорами ваг паралельно, використовуючи можливості не лише CPU, а й графічного процесору (GPU), що значно прискорює процес виконання Python-коду, що використовує Tensorflow [13].

Також при реалізації архітектури LODE-GRU, описаної у розділі 2, необхідно «загортати» ШНМ у архітектуру варіаційного автокодувальника, що передбачає виконання операцій над розподілами, та в цілому обробку ансамблю можливих прогнозованих значень ЧР шляхом взяття не фіксованої початкової точки для декодування значень ряду з прихованого простору, а вибору точки з деякого розподілу. Для таких цілей у Tensorflow версії 2 додана бібліотека Tensorflow Probability, що слугує для ймовірнісного моделювання та створення Байєсівських ШНМ.

Таким чином, поєднуючи простоту написання коду мовою Python та обчислювальні можливості Tensorflow та CUDA, можна зробити швидким не тільки процес написання програмного коду, а й його виконання.

3.2 Аналіз вимог користувача до програмної реалізації мережі

Одними з основних критеріїв якості продукту були зручність, швидкість та адекватність моделі. Пояснимо кожен з критеріїв.

Зручність полягає у тому, що кінцевий користувач має звести до мінімуму маніпуляції над вхідними даними, тобто оперуючи лише значеннями ряду, приведеними до одиничного інтервалу. При цьому не обов’язково приводити ряд до стаціонарного шляхом виключення тренду, експериментувати з шириною «вікна» ковзного середнього, кількості лагів, що враховуються моделлю для апроксимації досліджуваного процесу авторегресійним, тощо. Таким чином метрика зручності для даної моделі ототожнюється з малими часовими витратами на попередню обробку даних.

Наступний фактор – швидкодія моделі повинна забезпечуватися не лише обчислювальними можливостями ЕОМ, а й можливістю збереження стану системи, щоб не проводити процес оптимізації вагових коефіцієнтів кожного разу, а лише «довчити» модель на нових даних.

Вирішальний та найважливіший критерій – адекватність моделі. Він полягає у тому, що модель засвоїть закономірності у вхідних послідовностях та зможе відтворити поведінку даних, знаючи кілька вхідних значень та користуючись апроксимацією розподілу початкової точки даного ряду, подану у якості суміші розподілів точок змодельованих траєкторій у прихованих станах системи.

Отже, програмний додаток, що розроблюється, має відповідати вимогам зручності у користуванні та поданні вхідних даних, швидкості навчання, можливості збереження стану системи для подальшого використовування даної для інших даних з метою економії часу на «донавчання», а також, що найголовніше, продукувати адекватні прогнози на основі вхідних даних.

3.3 Аналіз архітектури системи моделювання динаміки часового ряду

Для моделювання ЧР методами, описаними в п.2 необхідно реалізувати методи розв’язання диференціальних рівнянь та створити архітектуру LODE-GRU мережі. Для даних цілей створимо наступні модулі:

- solvers.py – програмний модуль, в якому реалізовані базові абстрактні методи для інтегрування ЗДР.
- adams.py, fixed_adams.py – алгоритм Адамса чисельного розв’язку ДР з адаптивним та фіксованим кроком інтеграції відповідно.
- rk_common.py – реалізація явного алгоритму Рунге-Кутти 4-го порядку.
- dopri5.py – адаптивний явний алгоритм Рунге-Кутти 4(5) порядку (у зарубіжній літературі відомий під назвою Dormand–Prince method). Його особливість полягає у використанні метода Рунге-Кутти 4-го порядку для розв’язку ДР та метода РК 5-го порядку для визначення величини похибки [14].
- tsit5.py – адаптивний явний алгоритм Рунге-Кутти 5(4) порядку (у зарубіжній літературі іменується 5th order Tsitouras method). На відміну від попереднього алгоритму для розв’язку ДР використовується метод Рунге-Кутти 5-го порядку, а метод РК 4-го порядку - для визначення величини похибки [15].
- odeint.py – модуль, що реалізує функції інтеграції ДР згідно обраного методу.
- adjoint.py – програмний модуль, що реалізує алгоритм спряжених рівнянь, а саме інтегрує спряжені до вхідних диференціальні рівняння.
- losses.py – програмний модуль, у якому реалізовані функції похибок для мережі, а саме: логарифмічна правдоподібність для нормально розподілених випадкових величин та логарифмічна правдоподібність для нестаціонарного пуасонівського процесу.
- model.py – модуль, в якому реалізовані модифікації GRU-мережі, кодувальник LODE-GRU, декодувальник та baseline-модель варіаційного автокодувальника.

– ode_block.py – модуль, в якому реалізовані базові ЗДР-блоки для інтеграції в існуючі архітектури шарів нейронних мереж. Зокрема – це, насамперед, шар прямозв'язної ШНМ (Dense шар), що імітує систему диференціальних рівнянь; LatentODE модель, що інтегрується в рекурентну ШНМ через спеціальний клас – DiffeqSolver, котрий використовує методи інтегрування ЗДР для прямого розповсюдження даних крізь шар мережі. Ключовою системою є LatentODEVAE клас, що використовує архітектуру варіаційного автокодувальника з LODE-GRU мережею у якості кодувальника та одношарового Dense-декодувальника.

Тепер необхідно зобразити приблизну схему мережі. Як було обговорено у п. 2.4, у кінцевої мережі є дві модифікації – з моделюванням розподілу часових міток, та без нього. В обох випадках на вхід до мережі подається тензор даних разом з відповідними часовими мітками, але у другому випадку відразу приймається гіпотеза про рівномірний розподіл цих часових зрізів на інтервалі спостережень.

Зобразимо на рис. 3.1. тепер архітектуру кінцевої мережі, що є загальною для обох випадків:

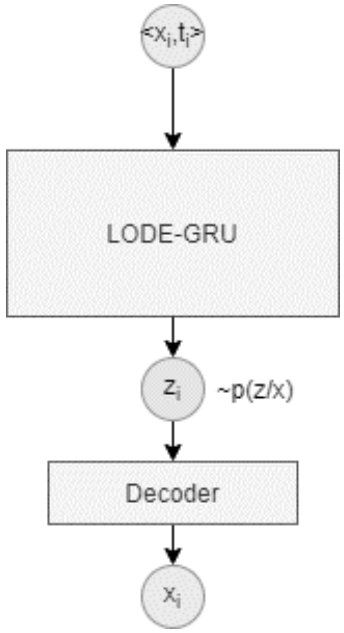


Рисунок 3.3 – Спрощена архітектура варіаційного автокодувальника

Далі, зобразимо архітектуру LODE-GRU кодувальника, що описується рівняннями (2.25)-(2.30) (рис. 3.2.):

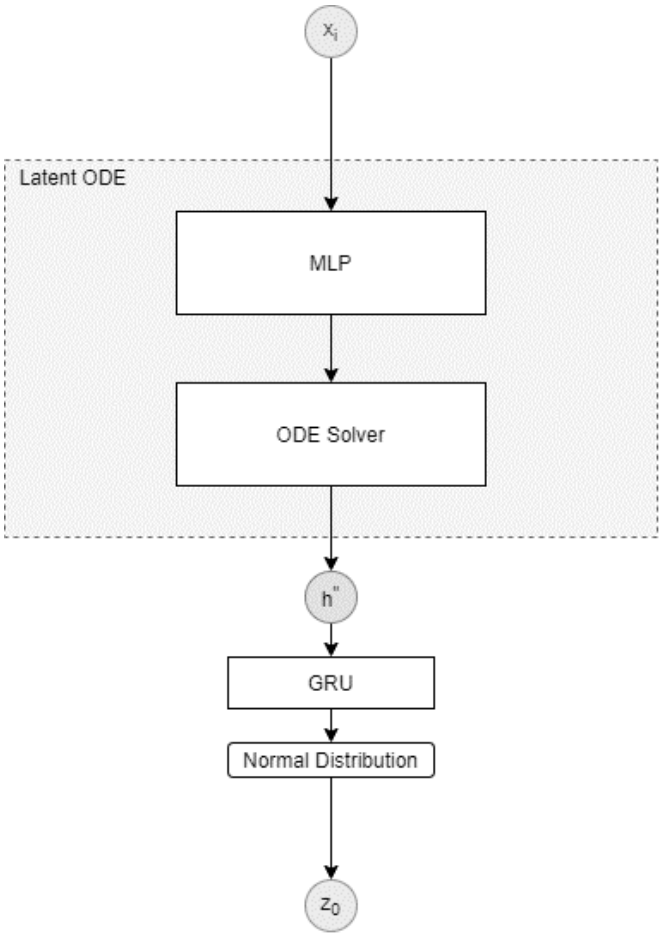


Рисунок 3.3 – Спрощена архітектура LODE-GRU кодувальника

Тут і надалі MLP – багатошаровий перцептрон. Тепер зобразимо на рис. 3.3. архітектуру декодувальника:

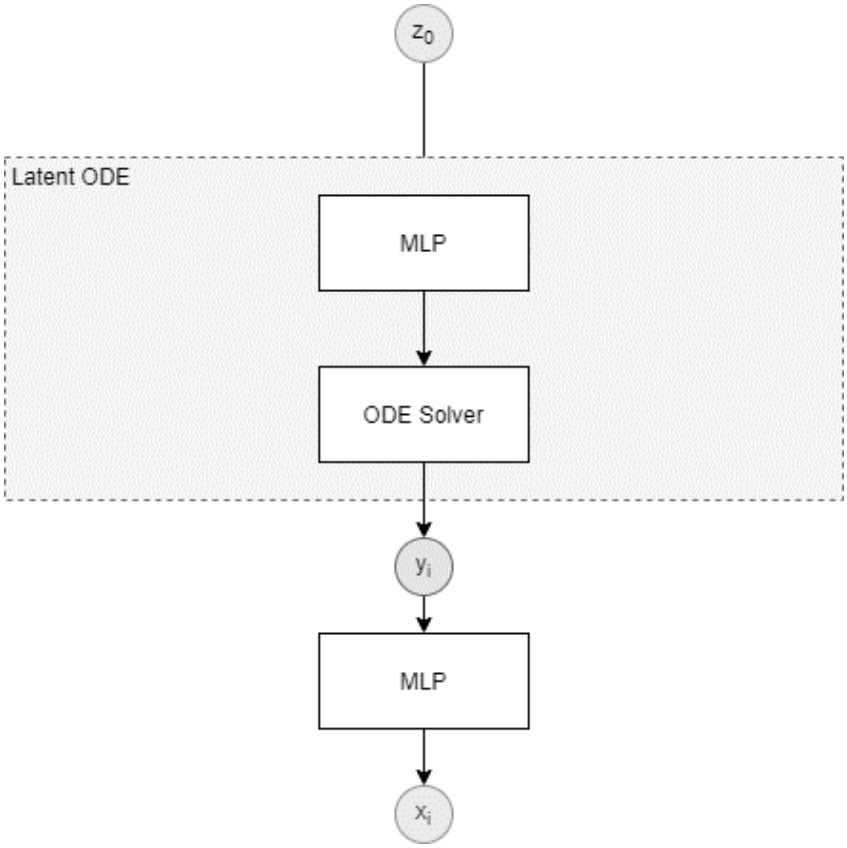


Рисунок 3.3 – Спрощена архітектура декодувальника

Єдиною відмінністю обох варіантів реалізації є те, що моделююча Latent ODE мережа при моделюванні нестаціонарних пуасонівських процесів приймає на вхід часові мітки разом з закодованими значеннями вхідних даних. При цьому Latent ODE мережа доповнюється ще однією MLP-мережею, що моделює цей пуасонівський процес та «повертає» значення параметру λ.

3.4 Постановка експерименту. Вибір гіперпараметрів моделі

Після опису архітектури цільової мережі перейдемо до опису експерименту з прогнозування часового ряду. Для експерименту візьмемо нестаціонарний синтетичний часовий ряд. Серед найпростіших випадків візьмемо ряд, що являє собою функцію

$$x = \sin(2\pi\omega t)$$

де $\omega \in \mathbb{R}, t \in [0, T]$. Даний ряд не є стаціонарним, оскільки в ньому присутній фактор сезонності.

Для того, щоб дані були більш наближеними до реальних, додамо до кожного значення ряду випадковий шум. Таким чином, отримаємо

$$x = \sin(\pi\omega t) + \epsilon$$

де $\epsilon \sim \mathcal{N}(0, \sigma), \sigma \in \mathbb{R}^+$.

Але, у реальних задачах прогнозування далеко не завжди доступні всі історичні дані. Тому введемо додатково величини d – крок дискретизації, та i – індикаторну множину точок, які треба залишити у вибірці. Отже, ряд, що моделюється, може бути формально записаний у наступному вигляді:

$$\hat{X} = X^T i = \|(t_k, \sin(\pi\omega t_k))\|^T{}_{k=\overline{1,d}} i \tag{3.1}$$

Далі необхідно обрати такі гіперпараметри мережі:

- Кількість шарів у мережі, що моделює поведінку ЗДР та їх розмірність.
- Коефіцієнт навчання.
- Розмірність прихованого простору.
- Метод розв’язку диференціального рівняння.
- Метод навчання мережі (тобто, оптимізатор).
- Початкове припущення про дисперсію значень вхідного ряду.
- Кількість епох навчання моделі.

Для експерименту були обрані такі параметри:

- $\omega = 0.25$.
- $d = 1000$.
- Кількість точок, що мають залишитися – 150.

- Кількість шарів у мережі, що моделює поведінку ЗДР та їх розмірність – 1 та 6 відповідно.
- Коефіцієнт навчання – адаптивний, спочатку рівний 0.01.
- Метод розв’язку диференціального рівняння - Dormand–Prince.
- Оптимізатор – Adamax, при цьому градієнти всередині ЗДР-шарів обраховуються методом спряжених рівнянь.
- Розмірність прихованого простору – 6.
- Початкове значення дисперсії – 0.1.
- Кількість епох – 200.

3.5 Аналіз та обробка результатів експерименту.

Після обрання гіперпараметрів, необхідно також визначитися з метриками, якими можна оцінити якість моделі. Цими метриками можуть являтися складові функції похибок, але вони не є універсальними показниками, та їх складно інтерпретувати (останнє твердження стосується дивергенції Куллбака-Лейблера). Тому потрібні «універсальні» метрики, у якості яких запропоновані середньоквадратична похибка та коефіцієнт детермінації R^2.

Зобразимо результати обчислень на наступних рисунках 3.4 – 3.6.

Спочатку зобразимо на рис.3.4. вхідні дані, отримані шляхом семплювання (3.1) на 1000 інтервалів, і виокремлення кожної 10-ї точки з кожного інтервалу.

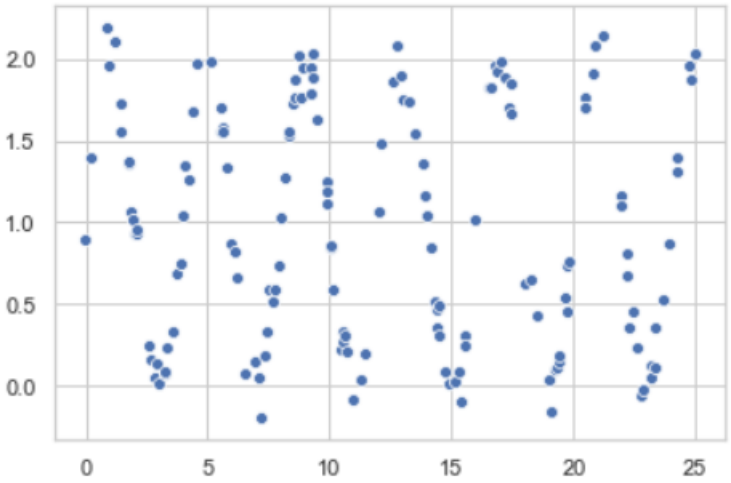


Рисунок 3.5 - Вхідні дані

Далі подамо на вхід до мережі точки з інтервалу $t = [0; 10)$ і проведемо навчання мережі. У якості тестової вибірки виступають ті точки, що залишилося. Результати реконструкції траєкторії наведені на рис. 3.5:

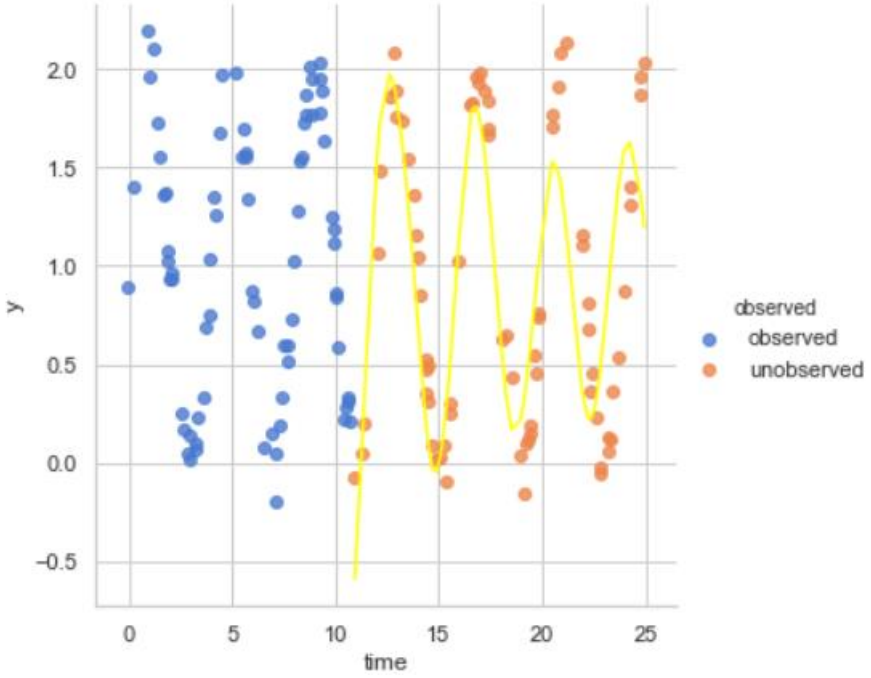


Рисунок 3.5 – Результати реконструкції. Синім позначені точки, що відповідають навчальній вибірці, зеленим – тестовій. Жовта крива – результат реконструкції

Як видно з рисунку вище, нейронна мережа даного типу змогла впоратися з нестаціонарністю досліджуваного процесу та вивчила головну особливість такого ряду – сезонність навіть попри непостійність вимірів.

На рисунку 3.6 зображений довгостроковий прогноз. Зеленою лінією позначені резудьтати роботи мережі з можливістю вивчення розподілу фіксацій значень ряду, червоною – мережа, що володіє припущенням про рівномірний розподіл моментів фіксації вимірів, а синім – істинна знешумлена траєкторія:

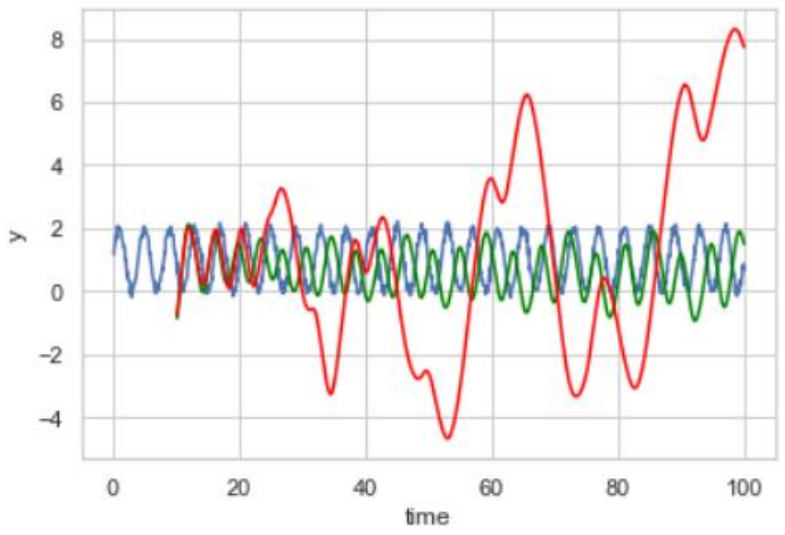


Рисунок 3.5 – Результати довгострокового прогнозу

Як бачимо, попри хороші результати на тестовій вибірці і в цілому засвоєнню обома моделями сезонності, у довгостроковій перспективі за рахунок малої кількості даних для тренування (50 точок з 1000) з плином часу в моделі зростає невизначеність. Особливо це виражається у спрощеній моделі.

Віднесемо до таблиці 3.1. показники метрик середньоквадратичної похибки MSE та коефіцієнту детермінації R^2. Дані метрики були обрані через свою універсальність. Також слід зазначити, що зовсім не обов'язково мати найкращі показники обох парметрів (0 для MSE та 1 для R^2), оскільки це може свідчити про перенавчання моделі (оскільки повністю зберігається дисперсія зашумленого процесу). У даній таблиці метрики MSE та R^2 будуть досліджені у розрізі кількості наявних вхідних даних.

Таблиця 3.1 – Матриця значень основних показників адекватності моделі

Найменування параметру	Значення параметру в залежності від розмірності відомих даних		
	10% від загальної к-сті	20% від загальної к-сті	50% від загальної к-сті
MSE	0,15	0,11	0,06
R^2	0,73	0,88	0,90

Отже, судячи по таблиці та графікам вище, можна зробити висновок, що дана модель відновлення динаміки ЧР справилася зі своєю задачею дуже добре, навіть попри нерівномірність вимірів та малої кількості вхідних даних, що часто трапляється і у реальному житті.

3.6 Висновки

У даному розділі були сформульовані та проаналізовані вимоги потенційних користувачів до програмної реалізації моделі, обґрунтований вибір мови програмування та додаткових програмних модулів, проаналізована архітектура досліджуваної системи та формально наведена постановка експерименту та його результати.

У п. 3.1. були обговорені можливі варіанти мов програмування, що будуть найбільш зручними для реалізації моделі. В якості такої була обрана мова програмування Python 3, основними аргументами на користь такого вибору є швидкість написання коду та популярність мови. Далі був визначений найбільш зручний фреймворк для спрощення реалізації моделі. Найкращим варіантом з точки зору використання ресурсів апаратного

забезпечення та швидкодії виявився Tensorflow 2, що використовує можливості бібліотеки Nvidia CUDA та має інтегровану бібліотеку Keras, що використовується даним фреймворком як високорівневий API для швидкого конструювання моделей глибинного навчання.

У наступному пункті були обговорені основні вимоги користувачі до програмного модуля. Серед основних критеріїв якості моделі були зручність, швидкість та адекватність моделі.

У п. 3.3. були наведені основні програмні модулі, що забезпечують реалізацію заданої нейронної мережі, а також зображена спрощена архітектура даної ШНМ.

Також в п. 3.4. була сформульована задача-експеримент, що демонструє працездатність та адекватність моделі та обрані основні метрики оцінки роботи моделі. Результати роботи моделі були обговорені у наступному пункті даного розділу.

4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ

У даному розділі буде проведено функціонально-вартісний аналіз спроектованого програмного додатку-реалізації моделі прогнозування та відтворення динаміки часового ряду методами штучних нейронних мереж. Сформулюємо задачу проектування такого програмного продукту.

4.1 Постановка задачі проектування

Основна задача проектування полягає у необхідності спроектувати програмний продукт-модель для аналізу нестаціонарного часового ряду, що була би зручною у користуванні та в цілому адекватною по відповідним метрикам якості (напр., дивергенції Куллбака-Лейблера, коефіцієнту детермінації, тощо).

Програмний продукт призначений для використання в середовищах що підтримують технології Tensorflow та на машинах, що мають додатково встановлену мову програмування Python.

4.2 Обґрунтування функцій та параметрів програмного продукту

Виходячи з конкретних цілей, які реалізуються, були сформовані наступні функції програмного продукту:

F1 – попередні маніпуляції над вимірюваним рядом:

а) приведення до одиничної шкали

б) повна попередня обробка даних, що включає в себе приведення ряду до стаціонарного

F2 – зберігання часових міток в даних:

а) подання на вхід до моделі пари (часова мітка, значення ряду)

б) використання лише даних

F3 – обробка ряду для побудови моделі

а) рекурентна нейронна мережа на основі LSTM

б) нейронна мережа на основі GRU

в) варіаційний автокодувальник на основі нейронних ЗДУ

г) ARMA/ARIMA/GARCH моделі

F4 – отримання результатів прогнозування:

а) семплювання зі змодельованого розподілу

б) прогнозування на фіксовану кількість кроків нейромережею

в) прогнозування на основі очікуваних результатів

F5– збереження результатів роботи:

а) запис результатів у сховище та вивід користувачу

б) збереження стану системи

в) лише виведення користувачу

Виходячи з представлених варіантів будуємо морфологічну карту, що представлена на рисунку 4.1.

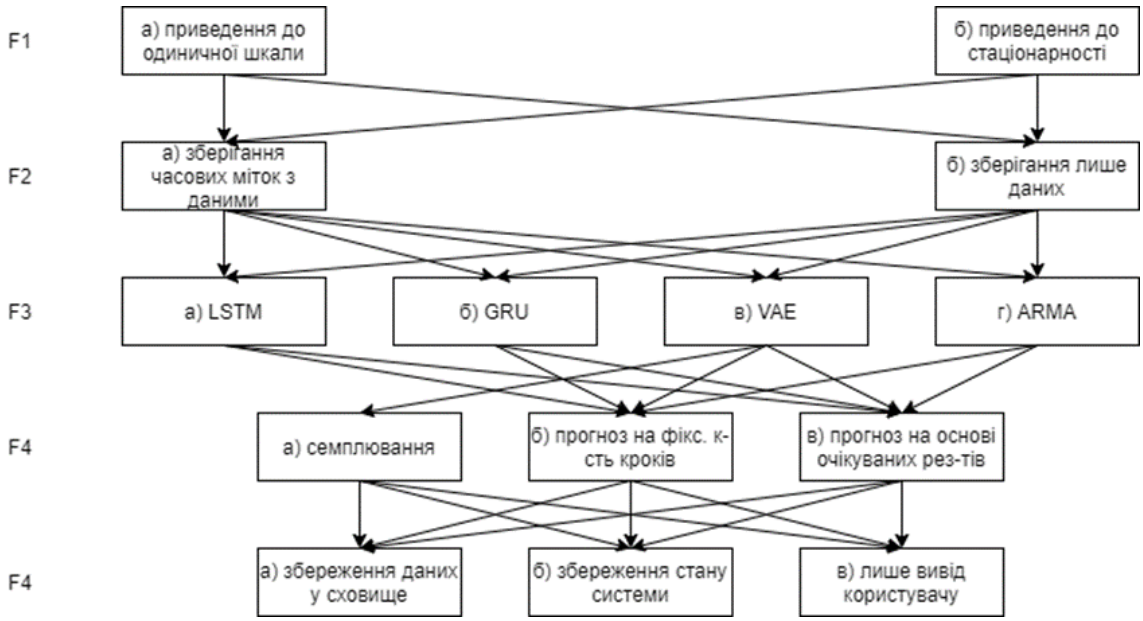


Рисунок 4.2 – Морфологічна карта

Спираючись на виконану карту необхідно побудувати позитивно-негативну матрицю. У даній матриці наведені переваги та недоліки для кожного варіанту реалізації кожного програмного модуля.

У таблиці 4.1 наведена позитивно-негативна матриця згідно морфологічної карти.

Таблиця 4.1 – Позитивно-негативна матриця

Основна функція	Варіант реалізації	Переваги	Недоліки
F1	А	Легкість у реалізації	Вимога до стаціонарності ряду, щоб отримана модель була адекватною
	Б	Широкий спектр засобів моделювання	Необхідно володіти деякими евристиками щодо даних
F2	А	Легкість реалізації	Ігнорується більшістю існуючих моделей
	Б	Легкість реалізації	Така варіація подання ігнорує природу розподілу часових міток
F3	А	Точність результатів	Складна реалізація, потреба в навчанні
	Б	Точність результатів	Складна реалізація, потреба в навчанні
	В	Високий рівень абстракції	Складна реалізація
	Г	Проста реалізація	Неточність прогнозів
F4	А	Висока точність	Вибірка може бути нерепрезентативною

Кінець таблиці 4.1

Основна функція	Варіант реалізації	Переваги	Недоліки
F4	Б	Висока точність	Неможливо отримати прогноз на довільну кількість кроків
	В	Зручність у використанні.	Накопичення похибки при прогнозуванні
F5	А	Можливість отримувати історичні дані	Складність реалізації
	Б	Можливість донавчання мережі	Необхідне потужне апаратне забзпечення
	В	Легкість реалізації	Недоступність попередніх результатів

Для характеристики прототипу програмного додатку використовуємо параметри X1 – X5.

На основі даних, що представлені у літературі, визначаємо мінімальні, середні отримуванні та максимально допустимі значення. Результати розрахунків для зручності представимо в якості таблиці (табл. 4.2).

Таблиця 4.2 – Система параметрів додатку

Найменування параметру	Позначення параметру	Значення параметру		
		Мінімальне	Середнє	Максимальне
Час розробки, людина*год	X1	352	528	704
Рекомендована частота процесору, ГГц	X2	2,3	2,8	4
Час навчання алгоритму, с	X3	100	3600	7200
Час обробки результату, с	X4	3	5	15
Рекомендована швидкість запису на диск, МБ/с	X5	32	64	128

Зобразимо можливі значення параметрів X1-X5 на рисунках нижче:

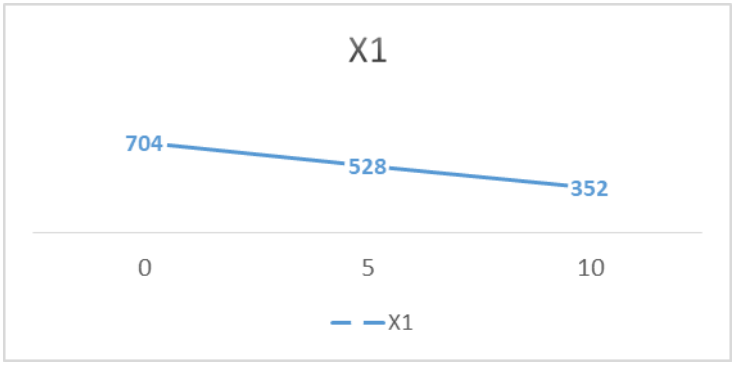


Рисунок 4.2 – Можливі значення X1

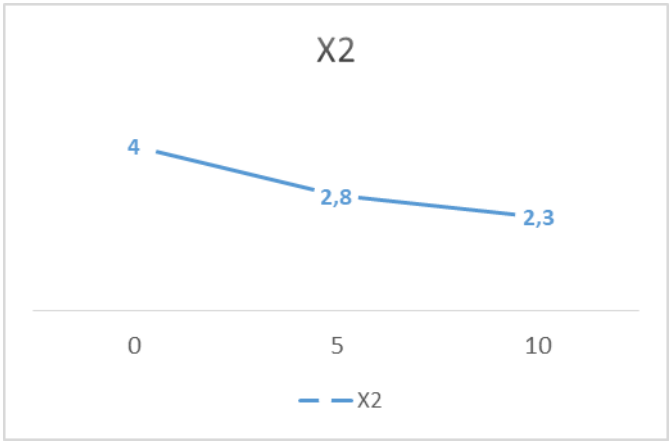


Рисунок 4.2 – Можливі значення X2

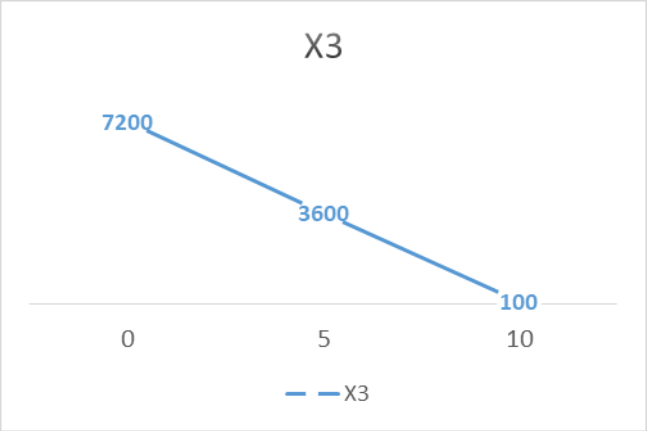


Рисунок 4.2 – Можливі значення X3

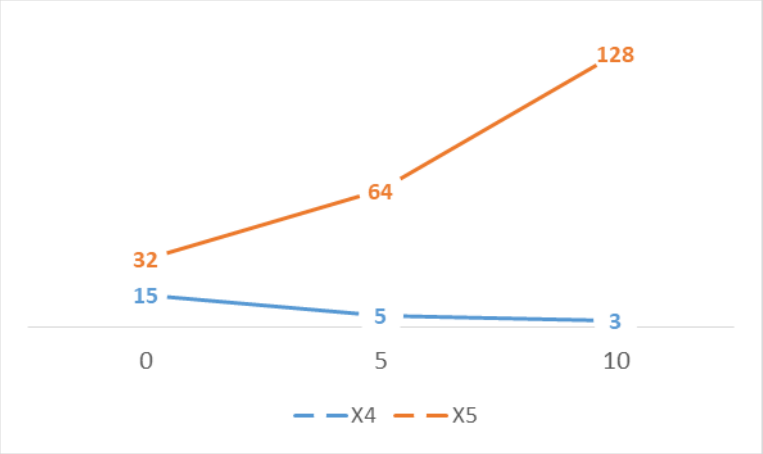


Рисунок 4.2 – Можливі значення X4 та X5

Вагомість параметрів оцінюється за допомогою методів попарного зрівняння. Ранги варіюються від 1 до 5. За найбільший ранг приймаємо 1.

Результати наведені в табл. 4.3-4.4

Таблиця 4.3 – Результат оцінки параметрів

Параметр	Ранг параметру по оцінці експерта							Сума рангів, R_i	ВідхиленняΔ_i	Квадрат відхилення, $(\Delta_i)^2$
	1	2	3	4	5	6	7			
X1	2	2	1	2	3	2	2	14	-7	49
X2	4	3	4	4	5	3	5	28	7	49
X3	1	1	2	1	2	1	1	9	-12	144
X4	3	4	3	3	1	4	4	22	1	1
X5	5	5	5	5	4	5	3	32	11	121

Кінець таблиці 4.3

Параметр	Ранг параметру по оцінці експерта							Сума рангів, R_i	ВідхиленняΔ_i	Квадрат відхилення, $(\Delta_i)^2$
	1	2	3	4	5	6	7			
Разом	15	15	15	15	15	15	15	105	0	364

Виконаємо попарне зрівнняння параметрів (табл. 4.4).

Таблиця 4.4 Попарне зрівняння параметрів

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 та X2	>	>	>	>	>	>	>	>	1.5
X1 та X3	<	<	>	<	<	<	<	<	0.5
X1 та X4	>	>	>	>	<	>	>	>	1.5
X1 та X5	>	>	>	>	>	>	>	>	1.5
X2 та X3	<	<	<	<	<	<	<	<	0.5
X2 та X4	<	>	<	<	<	>	<	<	0.5
X2 та X5	>	>	>	>	<	>	<	>	1.5
X3 та X4	>	>	>	>	<	>	>	<	1.5
X3 та X5	>	>	>	>	<	>	>	>	1.5
X4 та X5	>	>	>	>	>	>	<	>	1.5

Визначимо коефіцієнт конкордації:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 * 364}{7^2(5^3 - 5)} = 0.74 > W_k = 0.67$$

Так як коефіцієнт конкордації більше нормативного, результати вважають достовірними.

Розрахунок вагомості параметрів наведено в табл. 4.5:

Таблиця 4.5 – Розрахунок вагомості параметрів

Параметри	Параметри x_j					Перший крок		Другий крок		Третій крок	
	X1	X2	X3	X4	X5	b_i	$К_{вi}$	b_i	$К_{вi}$	b_i	$К_{вi}$
X1	1	1,5	0,5	1,5	1,5	6	0,24	27,5	0,24	124,75	0,24
X2	0,5	1	0,5	0,5	1,5	4	0,16	17,5	0,15	80,25	0,15
X3	1,5	1,5	1	1,5	1,5	7	0,28	34	0,30	155,50	0,30
X4	0,5	1,5	0,5	1	1,5	5	0,2	22	0,19	100	0,19
X5	0,5	0,5	0,5	0,5	1	3	0,12	14	0,12	64,50	0,12
Загалом:						25	1	115	1	525	1,00

Враховуючи дані з порівнянь варіантів реалізацій функцій можна виключити з реалізацій функцій наступні варіанти: F3(а, б), F4(б, в), F5(а, б). Залишаються наступні варіанти:

- F1(а)=>F2(а)=>F3(в)=>F4(а)=>F5(в)
- F1(б)=>F2(а)=>F3(в)=>F4(а)=>F5(в)
- F1(а)=>F2(б)=>F3(в)=>F4(а)=>F5(в)
- F1(б)=>F2(б)=>F3(в)=>F4(а)=>F5(в)
- F1(а)=>F2(а)=>F3(г)=>F4(а)=>F5(в)
- F1(б)=>F2(а)=>F3(г)=>F4(а)=>F5(в)
- F1(а)=>F2(б)=>F3(г)=>F4(а)=>F5(в)
- F1(б)=>F2(б)=>F3(г)=>F4(а)=>F5(в)

Виконаємо розрахунок вагомостей параметрів X у розрізі можливих значень цих параметрів, щоб отримати коефіцієнт якості реалізації кожної функції.

Результати розрахунків для зручності представлені таблицею 4.6.

Таблиця 4.6 – Розрахунок вагомості параметрів

Основна функція	Варіант реалізації	Абсолютне значення параметру	Бальна оцінка параметру	Коефіцієнт вагомості параметру	Коефіцієнт якості
F1(X1)	а)	400	6,01	0,24	1,44
	б)	600	3,66	0,24	0,88
F2(X2)	а)	3	4,8	0,15	0,72
	б)	2,8	5	0,15	0,75
F3(X3)	в)	2400	7,22	0,3	2,16
	г)	120	9,7	0,3	2,91
F4(X4)	а)	5	5	0,19	0,95
F5(X5)	в)	86	6,72	0,12	0,81

Обрахуємо коефіцієнти якості кожного з варіантів розробки:

$$K_{я1} = 1{,}44 + 0{,}72 + 2{,}16 + 0{,}95 + 0{,}81 = 6{,}08,$$
$$K_{я2} = 0{,}88 + 0{,}72 + 2{,}16 + 0{,}95 + 0{,}81 = 5{,}52,$$
$$K_{я3} = 1{,}44 + 0{,}75 + 2{,}16 + 0{,}95 + 0{,}81 = 6{,}11,$$
$$K_{я4} = 0{,}88 + 0{,}75 + 2{,}16 + 0{,}95 + 0{,}81 = 5{,}55,$$
$$K_{я5} = 1{,}44 + 0{,}72 + 2{,}91 + 0{,}95 + 0{,}81 = 6{,}83,$$
$$K_{я6} = 0{,}88 + 0{,}72 + 2{,}91 + 0{,}95 + 0{,}81 = 6{,}27,$$
$$K_{я7} = 1{,}44 + 0{,}75 + 2{,}91 + 0{,}95 + 0{,}81 = 6{,}86,$$
$$K_{я8} = 0{,}88 + 0{,}75 + 2{,}91 + 0{,}95 + 0{,}81 = 6{,}3$$

Оскільки варіант 7 має найбільший коефіцієнт якості, він є найкращим.

4.3 Економічний аналіз варіантів розробки

Для оцінки трудомісткості розробки спочатку проведемо розрахунок трудомісткості. Усі варіантимають два основних завдання – семплювання даних зі змодельованого розподілу та вивід результатів обчислень на екран.

Також кожний з варіантів має два додаткових завдання, які є реалізаціями розгалужених варіантів розробки незалежного модуля. Далі наведено варіанти додаткових завдань:

1.1) Приведення вхідних даних до одиничної шкали

1.2) Повна попередня обробка даних

2.1) Подання на вхід до моделі пари (часова мітка, значення ряду)

2.2) Використання лише даних, без прив’язки часової мітки

3.1) Використання варіаційнихавтокодувальників на основінейронних ЗДР

3.2) Використання «класичних» моделей аналізу часових рядів

У варіанті 1 присутні наступні додаткові завдання під номерами 1.1, 2.1 та 3.1

У варіанті 2 присутні наступні додаткові завдання під номерами 1.2, 2.1 та 3.1

У варіанті 3 присутні наступні додаткові завдання під номерами 1.1, 2.2 та 3.1

У варіанті 4 присутні наступні додаткові завдання під номерами 1.2, 2.2 та 3.1

У варіанті 5 присутні наступні додаткові завдання під номерами 1.1, 2.1 та 3.2

У варіанті 6 присутні наступні додаткові завдання під номерами 1.2, 2.1 та 3.2

У варіанті 7 присутні наступні додаткові завдання під номерами 1.1, 2.2 та 3.2

У варіанті 8 присутні наступні додаткові завдання під номерами 1.2, 2.2 та 3.2

За ступенем новизни до групи А відноситься задання 3.1, 3.2, до групи Б – завдання 1.1, 1.2, 2.1, 2.2. До групи В віднесемо завдання 4 та 5.

За складністю алгоритмів до групи 1 відносяться завдання 1.2, 3.1, 3.2, 2.1.До групи 3 відноситься завдання 1.1, 2.2, 4, 5.

Спираючись на норми розрахункового часу визначимо трудомісткість. Вона складає для завданнь 3.1 та 1.2 $T_р$=90 людино-днів. Поправочний коефіцієнт складає $К_п$=1,7. Оскільки під час виконання даного завдання використовуються новостворені модулі, врахуємо це за допомогою коефіцієнта $К_{ст}$ = 0,6. Коефіцієнти $К_м$ і $К_{ст.п}$, які враховують відповідно програмування на мові низького рівня та розробку стандартного програмного забезпечення, для всіх семи завдань дорівнюють 1.

Повна трудомісткість завдання 3.1 (складність – 1, новизна – А):

$$T_1 = 90 * 1{,}7 * 0{,}6 = 91{,}8$$

Аналогічно для завдання 1.2 (складність – 1, новизна – Б), де $T_р$= 64; $К_п$ = 1,02; $К_{ст}$ = 0,8:

$$T_2 = 64 * 1{,}02 * 0{,}8 = 52{,}22$$

Аналогічно для завдання 3.2 (складність – 1, новизна – А), де $T_р$= 90; $К_п$ = 1,7; $К_{ст}$ = 0,8:

$$T_3 = 90 * 1{,}7 * 0{,}8 = 122{,}4$$

Для завдання 1.1 (складність – 3, новизна – Б) $T_р$= 19; $К_п$ = 0,9; $К_{ст}$ = 0,6:

$$T_4 = 19 * 0{,}9 * 0{,}6 = 10{,}26$$

Для завдання 2.1 (складність – 1, новизна – Б) $T_р$= 64; $К_п$ = 1,02; $К_{ст}$ = 0,8:

$$T_5 = 64 * 1{,}02 * 0{,}8 = 52{,}22$$

Аналогічно для завдання 2.2 (складність – 3, новизна – Б) $T_р$= 19; $К_п$ = 0,9; $К_{ст}$ = 0,6:

$$T_6 = 19 * 0{,}9 * 0{,}6 = 10{,}26$$

Аналогічно для завданнь 4 та 5 (складність – 3, новизна – В) $T_р$= 12; $К_п$ = 0,6; $К_{ст}$ = 0,8:

$$T_6 = 12 * 0{,}6 * 0{,}8 = 5{,}76$$

Визначимо повну трудомісткість варіантів (людино-днів):

$$T_1 = 10{,}26 + 52{,}22 + 91{,}8 + 5{,}76 + 5{,}76 = 165{,}8,$$
$$T_2 = 52{,}22 + 52{,}22 + 91{,}8 + 5{,}76 + 5{,}76 = 207{,}76,$$
$$T_3 = 10{,}26 + 10{,}26 + 91{,}8 + 5{,}76 + 5{,}76 = 123{,}84,$$
$$T_4 = 52{,}22 + 10{,}26 + 91{,}8 + 5{,}76 + 5{,}76 = 165{,}8,$$
$$T_5 = 10{,}26 + 52{,}22 + 122{,}4 + 5{,}76 + 5{,}76 = 196{,}4,$$
$$T_6 = 52{,}22 + 52{,}22 + 122{,}4 + 5{,}76 + 5{,}76 = 238{,}36,$$
$$T_7 = 10{,}26 + 10{,}26 + 122{,}4 + 5{,}76 + 5{,}76 = 154{,}44,$$
$$T_8 = 52{,}22 + 10{,}26 + 122{,}4 + 5{,}76 + 5{,}76 = 196{,}4$$

Найбільш трудомісткими завданням є 3.2, найбільш трудомісткий варіант – 6.

Далі вважається, що робочий день складає 8 годин, в тижні п’ять робочих днів. В розробці бере участь один програміст з окладом 35000 грн та

інженер з обробки даних з окладом 20000 грн. Визначимо середню заробітну плату за годину (грн):

$$С_ч = \frac{35000 + 20000}{2 * 22 * 8} = 156,25$$

Тоді заробітна плата для кожного з варіантів реалізації (грн):

1) $С_{ЗП} = 156,25 * 8 * 165,8 = 207250$
2) $С_{ЗП} = 156,25 * 8 * 207,76 = 259700$
3) $С_{ЗП} = 156,25 * 8 * 123,84 = 154800$
4) $С_{ЗП} = 156,25 * 8 * 165,8 = 207250$
5) $С_{ЗП} = 156,25 * 8 * 196,4 = 245500$
6) $С_{ЗП} = 156,25 * 8 * 238,36 = 297950$
7) $С_{ЗП} = 156,25 * 8 * 154,44 = 193050$
8) $С_{ЗП} = 156,25 * 8 * 196,4 = 245500$

Відрахування на соціальне страхування (22%) (грн):

1) $С_{ВІД} = 207250 * 0,22 = 55539$
2) $С_{ВІД} = 259700 * 0,22 = 56529$
3) $С_{ВІД} = 154800 * 0,22 = 69674$
4) $С_{ВІД} = 207250 * 0,22 = 55539$
5) $С_{ВІД} = 245500 * 0,22 = 69674$
6) $С_{ВІД} = 297950 * 0,22 = 77319$
7) $С_{ВІД} = 193050 * 0,22 = 50589$
8) $С_{ВІД} = 245500 * 0,22 = 69674$

Далі розрахуємо витрати на оплату однієї машино-години. Враховуючи, що вона обслуговує одного спеціаліста з окладом 35000 грн та одного з окладом 20000 грн з коефіцієнтом зайнятості 0,6, то для двох машин отримаємо (у год):

$$С_г = 12 * 35000 * 0,6 + 12 * 20000 * 0,6 = 396000$$

Враховуючи додаткову заробітну плату (грн):

$$С_{ЗП} = 396000 * (1 + 0,4) = 554400$$

Відрахування на соціальне страхування (грн):

$$С_{ВІД} = 554400 * 0,22 = 121968$$

Розрахуємо амортизаційні підрахунки (амортизація 25%, вартість ЕОМ 45000 грн)

$$С_А = К_{ТМ} * К_А * Ц_{ПР} = 1,15 * 0,25 * 45000 = 12937,5$$

Розрахуємо витрати на ремонт та профілактику (грн):

$$С_Р = К_{ТМ} * Ц_{ПР} * К_Р = 1,15 * 45000 * 0,05 = 2587,5$$

Розрахуємо ефективний годинний фонд часу ПК за рік

$$Т_{ЕФ} = (365 - 142 - 16) * 8 * 0,8 = 1324,8$$

Розрахуємо витрати на електроенергію

$$С_{ЕЛ} = 1324,8 * 0,6 * 0,6 * 1,75 = 834,62$$

Накладні витрати рівні (грн):

$$С_Н = 45000 * 0{,}67 = 30150$$

Отже експлуатаційні витрати (грн):

$$С_{ЕКС} = 554400 + 121968 + 12937{,}5 + 2587{,}5 + 834{,}62 + 30150 = 722877{,}62$$

Тоді собівартість однієї машино-години ЕОМ дорівнюватиме:

$$С_{М-Г} = \frac{722877{,}62}{1324{,}8} = 545{,}67$$

Враховуючи, що всі роботи ведуться на ЕОМ, витрати на оплату машинного часу:

1) $С_{ЗП} = 545{,}67 * 8 * 165{,}8 = 725103{,}09$
2) $С_{ЗП} = 545{,}67 * 8 * 207{,}76 = 908609{,}27$
3) $С_{ЗП} = 545{,}67 * 8 * 123{,}84 = 541596{,}90$
4) $С_{ЗП} = 545{,}67 * 8 * 165{,}8 = 725103{,}09$
5) $С_{ЗП} = 545{,}67 * 8 * 196{,}4 = 858927{,}90$
6) $С_{ЗП} = 545{,}67 * 8 * 238{,}36 = 1042434{,}09$
7) $С_{ЗП} = 545{,}67 * 8 * 154{,}44 = 675421{,}72$
8) $С_{ЗП} = 545{,}67 * 8 * 196{,}4 = 858927{,}90$

Накладні витрати відповідно

1) $С_Н = 725103{,}09 * 0{,}67 = 485819{,}07$
2) $С_Н = 908609{,}27 * 0{,}67 = 608768{,}21$
3) $С_Н = 541596{,}90 * 0{,}67 = 362869{,}92$
4) $С_Н = 725103{,}09 * 0{,}67 = 485819{,}07$
5) $С_Н = 858927{,}90 * 0{,}67 = 575481{,}69$
6) $С_Н = 1042434{,}09 * 0{,}67 = 698430{,}84$
7) $С_Н = 675421{,}72 * 0{,}67 = 452532{,}55$

8) $С_Н = 858927{,}90 * 0{,}67 = 575481{,}69$

Розрахуємо повну вартість розробки за варіантами:

1) $С_{ПП} = 207250 * 1{,}22 + 725103{,}09 * 1{,}67 = 1463767{,}16$
2) $С_{ПП} = 259700 * 1{,}22 + 908609{,}27 * 1{,}67 = 1834211{,}49$
3) $С_{ПП} = 154800 * 1{,}22 + 541596{,}90 * 1{,}67 = 1093322{,}82$
4) $С_{ПП} = 207250 * 1{,}22 + 725103{,}09 * 1{,}67 = 1463767{,}16$
5) $С_{ПП} = 245500 * 1{,}22 + 858927{,}90 * 1{,}67 = 1733919{,}59$
6) $С_{ПП} = 297950 * 1{,}22 + 1042434{,}09 * 1{,}67 = 2104363{,}93$
7) $С_{ПП} = 193050 * 1{,}22 + 675421{,}72 * 1{,}67 = 1363475{,}27$
8) $С_{ПП} = 245500 * 1{,}22 + 858927{,}90 * 1{,}67 = 1733919{,}59$

4.4 Вибір кращого варіанта ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня

$$К_{ТЕР1} = \frac{6{,}08}{1463767{,}16} = 4{,}13 * 10^{-6}$$

$$К_{ТЕР2} = \frac{5{,}52}{1834211{,}49} = 3{,}01 * 10^{-6}$$

$$К_{ТЕР3} = \frac{6{,}11}{1093322{,}82} = 5{,}58 * 10^{-6}$$

$$К_{ТЕР4} = \frac{5{,}55}{1463767{,}16} = 3{,}8 * 10^{-6}$$

$$К_{ТЕР5} = \frac{6{,}83}{1733919{,}59} = 3{,}93 * 10^{-6}$$

$$К_{ТЕР6} = \frac{6{,}27}{2104363{,}93} = 2{,}98 * 10^{-6}$$

$$К_{ТЕР7} = \frac{6{,}86}{1363475{,}27} = 5{,}03 * 10^{-6}$$

$$К_{ТЕР8} = \frac{6{,}31}{1733919{,}59} = 3{,}64 * 10^{-6}$$

4.5 Висновки

Отже враховуючи всі дослідження, що описані вище, можна сказати, що 3 варіант реалізації є найбільш оптимальним зі сторони якісно-економічної оцінки. Його коефіцієнт техніко-економічного рівня складає $5{,}58 * 10^{-6}$.

Розробка цього варіанту передбачає такі обов’язкові завдання як:

- Семплювання даних зі змодельованого розподілу при прогнозі
- Вивід результатів обчислень на екран
- Серед завдань між якими ставився вибір в даному варіанті реалізовані такі завдання:
- Приведення даних до одиничного інтервалу
- Використання варіаційних автокодувальників для моделювання ряду
- Подання на вхід до моделі значення часового ряду без супроводжуючих часових міток

ВИСНОВКИ

Дана дипломна робота є дослідженням на тему застосування апарату диференціальних рівнянь до ШНМ у задачах відтворення динаміки часового ряду. Основною задачею даної роботи було розробити модель на основі ШНМ, що могла би використовуватись для моделювання динаміки досліджуваного процесу.

В роботі проведені наступні дослідження:

- Досліджені «класичні» методи аналізу часових рядів, такі як ARMA, ARIMA та GARCH.
- Досліджені методи машинного та глибинного навчання, що застосовуються для обробки послідовностей. Сформована задача прогнозування значень послідовності як задача навчання з вчителем.
- Проаналізовані породжуючі моделі глибинного навчання та моделі типу кодувальник-декодувальник для виявлення прихованих закономірностей у даних.
- Досліджені методи моделювання динаміки часових рядів шляхом використання апарату ЗДР та рекурентних нейронних мереж. Формалізована поведінка нейронних ЗДР-систем
- Програмно реалізована система типу кодувальник-декодувальник, що використовує ЗДР-мережі. Проведено експеримент, що доводить працездатність даної системи у задачах моделювання процесів, що залежать від неперервного часу.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Grandell J. Time series analysis. URL: https://www.math.kth.se/matstat/gru/sf2943/ts.pdf (Last acessed 01.04.2020).

2. Бідюк П. І., Романенко В. Д., Тимощук О. Л. Аналіз часових рядів : навчальний посібник. Київ : Політехніка, 2010. 317 с.

3. Николенко С., Кадурин А., Архангельская Е. Глубокое обучение. — Санкт-Петербург : Питер, 2018. 480 с.

4. Гудфеллоу Я., Бенджио И., Курвилль А. Глубокое обучение / пер. с англ. А. А. Слинкина. – 2-е изд., испр. Москва : ДМК Пресс, 2018. 652 с.

5. Паттаняк С. Глубокое обучение и Tensorflow для профессионалов. Математический подход к построению систем исскуственного интеллекта на Python. : пер. с англ. Санкт-Петербург. : Диалектика, 2020. 480 с.

6. Васильева А. Б., Медведев Г. Н., Тихонов Н. А., Уразгильдина Т. А. Дифференциальные и интегральные уравнения, вариационное исчисление в примерах и задачах. Москва : ФИЗМАТЛИТ, 2003. 432 с.

7. Понтрягин Л. С. Математическая теория оптимальных процессов и дифференциальные игры. *Топология, обыкновенные дифференциальные уравнения, динамические системы* : сб. обз. ст. 2. к 50-летию института, Тр. МИАН СССР./ ред. С. М. Никольский, Е. Ф. Мищенко, 1985. № 169. С.119–158.

8. Bradley A. M. PDE-constrained optimization and the adjoint method. URL: https://cs.stanford.edu/~ambrad/adjoint_tutorial.pdf

9. Chen R. T. Q., Rubanova Y., Bettencourt J., Duvenaud D. Neural Ordinary Differential Equations. URL: https://arxiv.org/pdf/1806.07366.pdf

10. Chen R. T. Q., Rubanova Y., Duvenaud D. Latent ODE for Irregularly-Sampled Time Series. URL: https://arxiv.org/pdf/1907.03907.pdf (Last acessed 27.04.2020).

11. Харрисон М. Как устроен Python. Гид для разработчиков, программистов и интересующихся. / пер. с англ. Е. Матвеева. Санкт-Петербург : Питер, 2019. 272 с.

12. TensorFlow, PyTorch or MXNet? A comprehensive evaluation on NLP & CV tasks with Titan RTX. *Medium* : a web-site. URL: https://medium.com/syncedreview/tensorflow-pytorch-or-mxnet-a-comprehensive-evaluation-on-nlp-cv-tasks-with-titan-rtx-cdf816fc3935#:~:text=TensorFlow%2C%20PyTorch%2C%20and%20MXNet%20are,three%20frameworks%20with%20GPU%20support.&text=For%20example%2C%20TensorFlow%20training%20speed,is%2024%25%20faster%20than%20MXNet

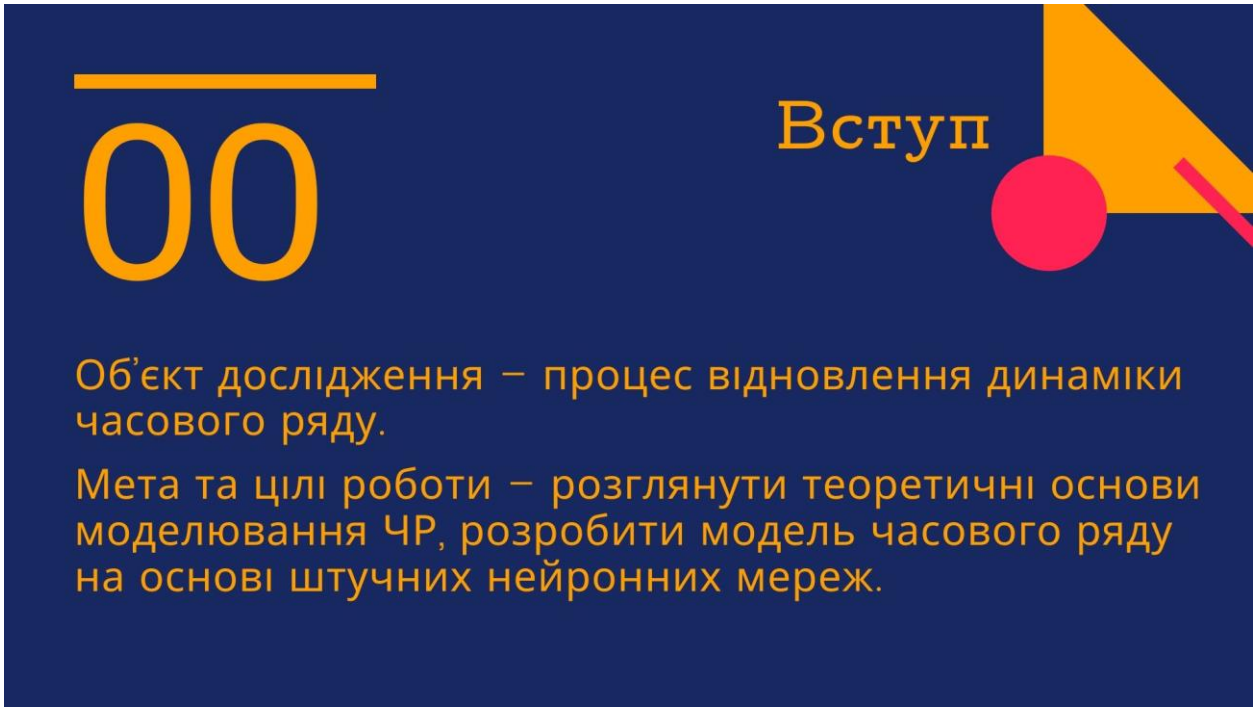
13. Introduction to TensorFlow. *Tensorflow* : a web-site. URL: https://www.tensorflow.org/learn

14. Cook J. D. A better adaptive Runge-Kutta method. *John D. Cook Conslting*: a web-site URL: https://www.johndcook.com/blog/2020/02/19/dormand-prince/

15. Ch. Tsitouras. Runge–Kutta pairs of orders 5(4) using the minimal set of simplifying assumptions. URL: https://www.researchgate.net/publication/251743368_Runge-Kutta_Pairs_of_Orders_54_using_the_Minimal_Set_of_Simplifying_Assumptions

ДОДАТОК А


Дипломний проект
Відтворення динаміки часового ряду методами ШНМ
ВИКОНАВ: АНДРОСОВ ДМИТРО, КА-65
НАУКОВИЙ КЕРІВНИК: ДОЦ НЕДАШКІВСЬКА Н.І.


00
Вступ
Об'єкт дослідження – процес відновлення динаміки часового ряду.
Мета та цілі роботи – розглянути теоретичні основи моделювання ЧР, розробити модель часового ряду на основі штучних нейронних мереж.

00

Вступ

Предмет дослідження – модель часового ряду на основі архітектури кодувальник-декодувальник.

01

Актуальність дослідження

Автоматизація обробки даних.
Побудова адекватних моделей нестаціонарних ЧР.
Аналіз часових рядів як процесів, що залежать від неперервного часу.


02
Аналіз існуючих моделей часових рядів


02
Авторегресивні моделі
Апроксимують досліджуваний процес як такий, що є лінійною комбінацією попередніх значень ряду.
До таких моделей відносять ARMA, ARIMA та GARCH.
Вищезазначені моделі потребують стаціонарності ряду, або використовують методи, що приводять ряд до такого.

02

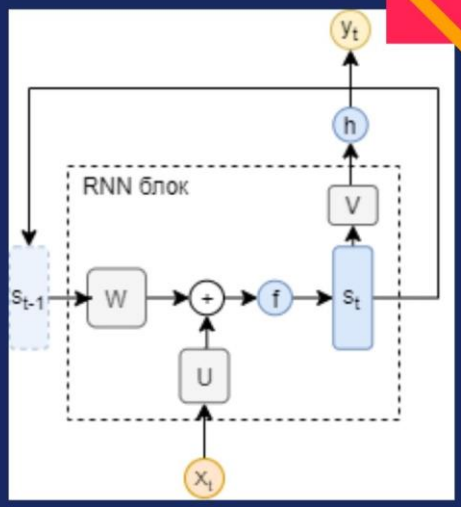
Моделі машинного навчання

До таких моделей відносять рекурентні нейронні мережі.

02

Рекурентні Нейронні Мережі

У Simple RNN на відміну від MLP зв'язки між нейронами можуть йти не тільки послідовно від попереднього шару до наступного, а й «до самого себе», точніше, до свого стану у попередній момент часу.



Архітектура RNN

W – матриця ваг-переходів між прихованими станами мережі
U – матриця ваг вхідного вектору
V – матриця ваг вихідного вектору

02

Рекурентні Нейронні Мережі

Недоліки:

- Неможливо навчити звичайними методами (Rprop)
- Проблема затухаючого градієнту
- Проблема вибуху градієнту

02

Рекурентні Нейронні Мережі

Навчання проводять алгоритмом BPTT:

- Градієнти обчислюються рекурсивно відносно часу
- Коригування ваг як в методі Rprop

02

Довга короткострокова пам'ять

Містить замість параметрів-констант Simple RNN механізм управляючих блоків, які імітують «пам'ять». Збільшена вкладеність мережі дозволяє вирішити проблему затухаючих градієнтів

Архітектура LSTM

На додачу до матриць ваг прихованих, вихідних та вхідних параметрів використовують управляючі конструкції – гейти

02

Довга короткострокова пам'ять

Недоліки:

- Складність навчання
- Громізкість архітектури

02

Рекурентні Вентильні Мережі

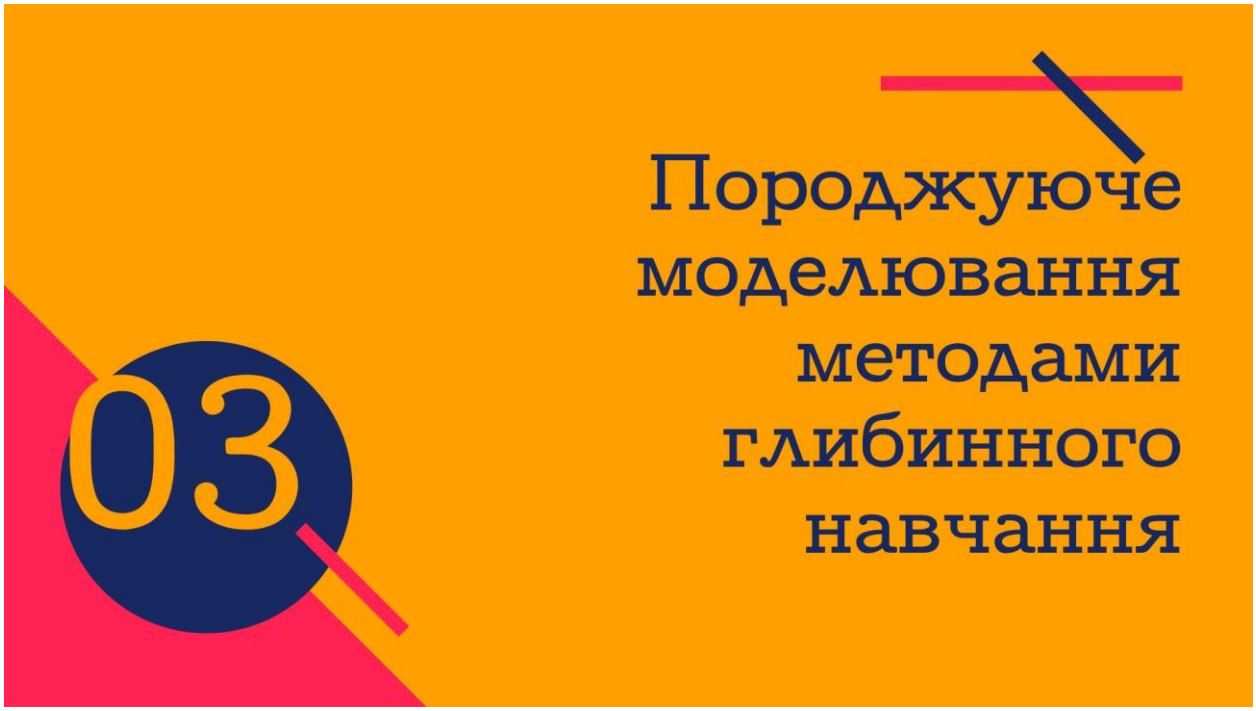
Детальний аналіз LSTM показав, що кількість її параметрів невиправдано велика.
Обєднання прихованого гейту та вихідного, введення гейту скидання та відкидання вихідного доозволяє скоротити кількість параметрів мережі.

02

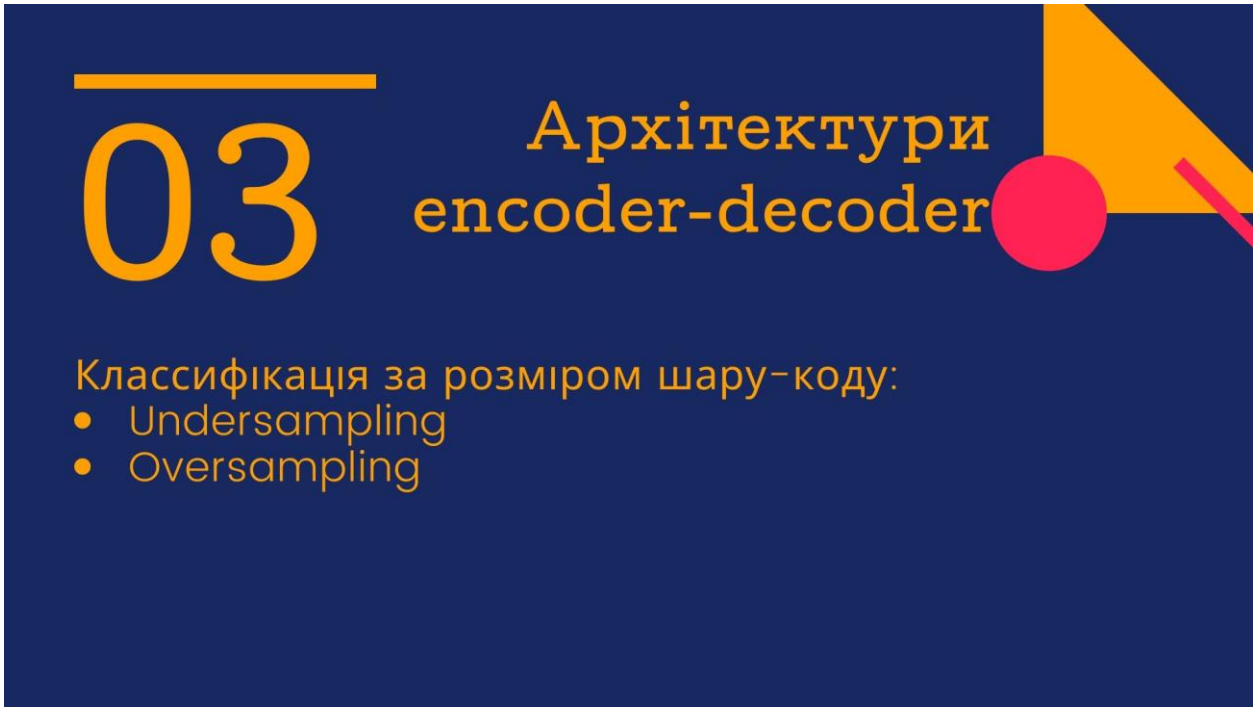
Довга короткострокова пам'ять

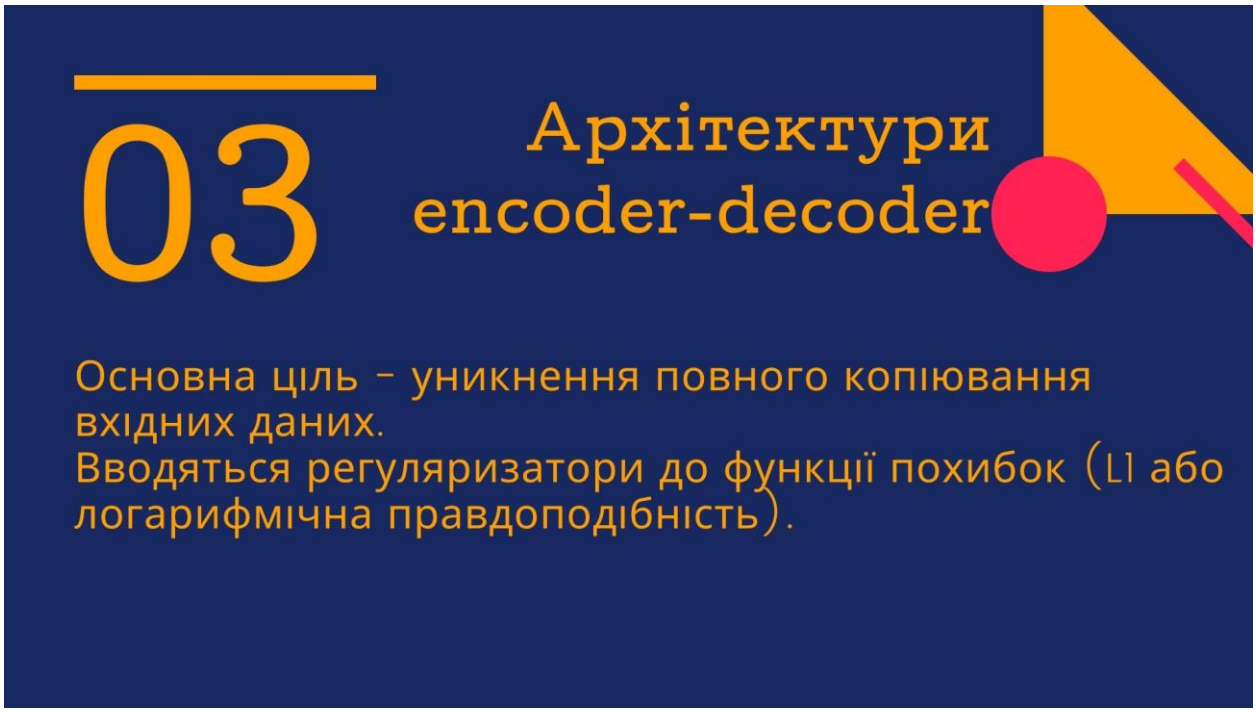
Недоліки:

- Необхідно вдало ініціювати початкові значення параметрів мережі, щоб вирішити проблему вибуху градієнту


03
Породжуюче моделювання методами глибинного навчання


03
Архітектури encoder-decoder
Причини використовувати:
• Виокремлення найбільш значущих ознак
• Пошук прихованих закономірностей


03
Архітектури
encoder-decoder
Классифікація за розміром шару-коду:
• Undersampling
• Oversampling


03
Архітектури
encoder-decoder
Основна ціль - уникнення повного копіювання вхідних даних.
Вводяться регуляризатори до функції похибок (L1 або логарифмічна правдоподібність).

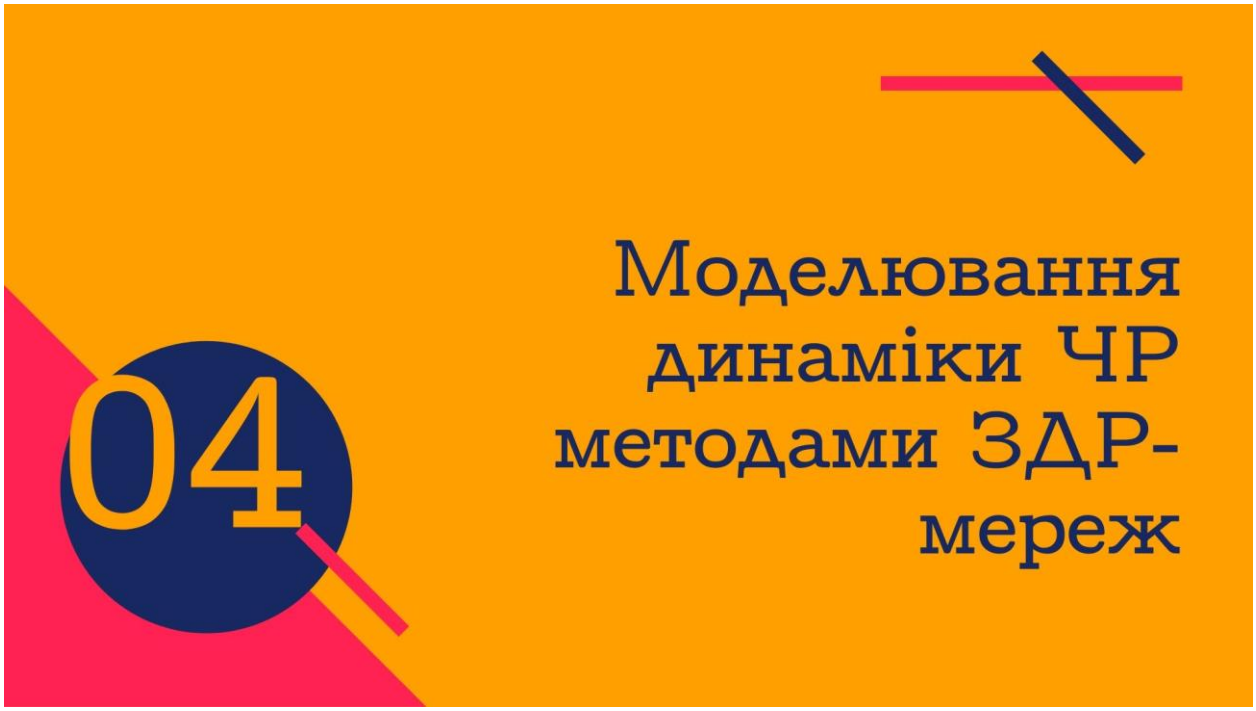
03 Variational Autoencoders

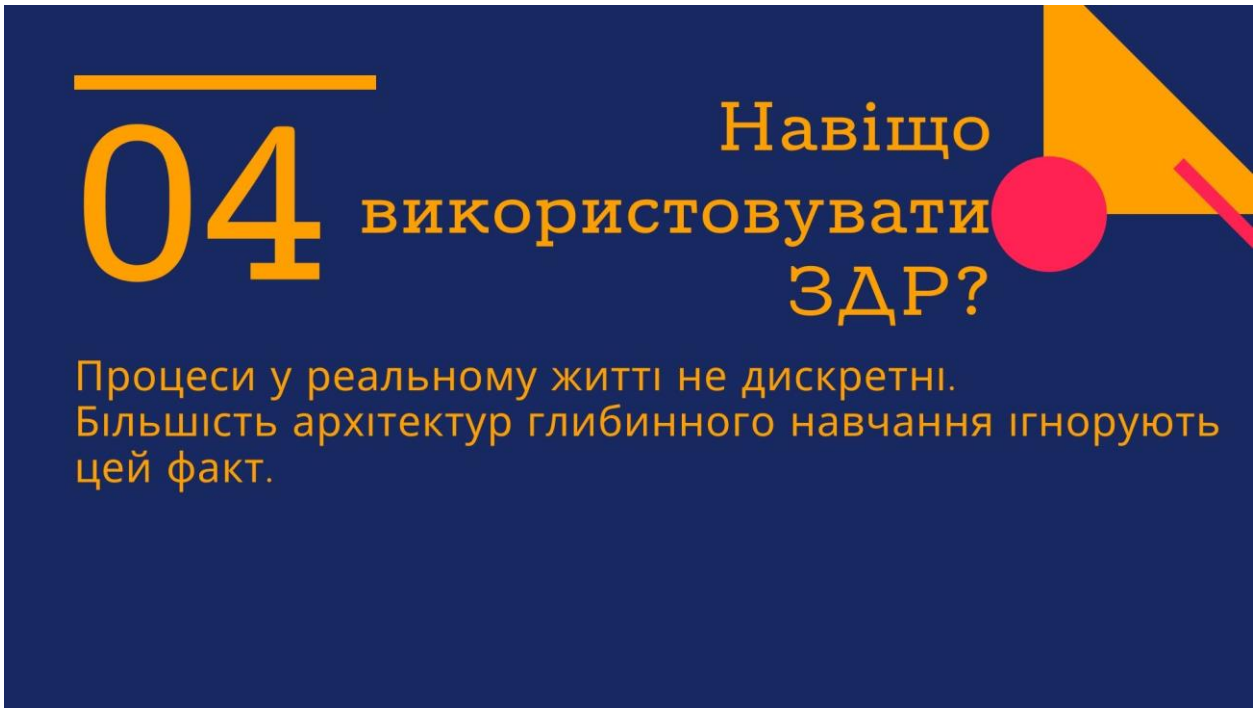
Простий код не дає достатньої описової повноти даних.
Замість константного коду – ймовірнісний розподіл.

03 Variational Autoencoders

Потрібна нестандартна функція похибок:

$$\mathcal{L}(q) = \mathbb{E}_{z\sim q(z/x)} \log p(x/z) - D_{KL}(q(z/x) \parallel p(z))$$


04
Моделювання динаміки ЧР методами ЗДР-мереж


04
Навіщо використовувати ЗДР?
Процеси у реальному житті не дискретні. Більшість архітектур глибинного навчання ігнорують цей факт.

04 Навіщо використовувати ЗДР?

У загальному випадку перетворення прихованих станів у РНМ мають вигляд:

$$h_{t+1} = h_t + f(h_t, \theta)$$

Зменшим крок дискретизації часу і отримаємо ДР:

$$\lim_{\Delta t \to 0} \frac{h_{t+\Delta t} - h_t}{\Delta t} = \frac{dh}{dt} = f(h, t, \theta)$$

04 Навіщо використовувати ЗДР?

Переваги підходу:

- Чутливість до часу-моментів фіксації значень
- Можливість доповнювати ряд новими значеннями
- Нові методи оптимізації параметрів моделі

04 Навіщо використовувати ЗДР?

Метод спряжених рівнянь (Понтрягін, 1962):

- Використовує спряжені диференційні рівняння для вирішення двоїстої задачі оптимізації з обмеженнями
- Дозволяє зекономити пам'ять при виконанні обчислень (Просторова складність O(1))

Метод спряжених рівнянь

ФУНКЦІЯ ПОХИБОК

$$L(z, t = t_1) = L(z(t_1)) = L\left(z(t_0) + \int_{t_0}^{t_1} f(z, t, \theta)dt\right)$$

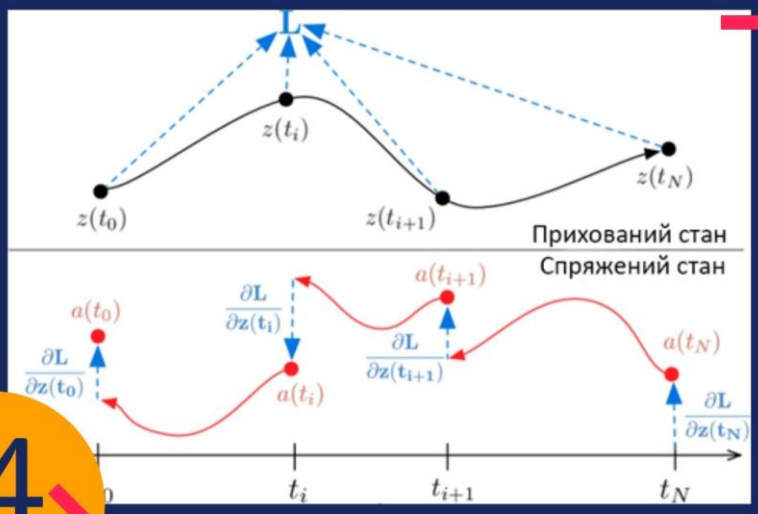
СПРЯЖЕНЕ РІВНЯННЯ-ПОХІДНА L ВІД ЧАСУ

$$\frac{da}{dt} = -a(t)\frac{df(z(t),t,\theta)}{dz(t)}$$

ГРАДІЄНТ L ВІДНОСНО ПАРАМЕТРІВ МОДЕЛІ

$$\frac{dL}{d\theta} = -\int_T^{t_0} a(t)\frac{\partial f(z(t),t,\theta)}{\partial \theta}dt$$

04



04

Графічна інтерпретація методу.

LODE-GRU мережі

ОСНОВА

Гібридна ШНМ, що використовує ЗДР-блоки для моделювання прихованого стану РНМ

НАВЧАННЯ

Навчаняя проводиться BPTT алгоритомом з використанням методу спряжених рівнянь

ЧУТЛИВІСТЬ ДО ЧАСУ

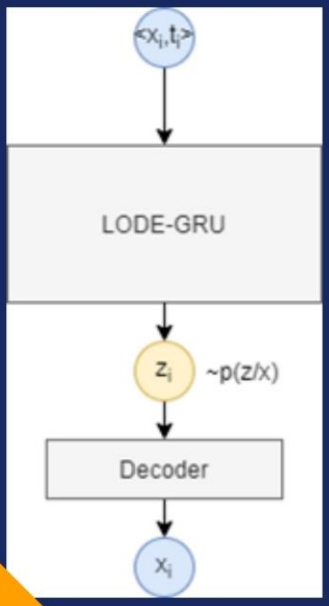
Використання нестаціонарних пуасонівських процесів для вивчення закону розподілу точок у часі

04

04

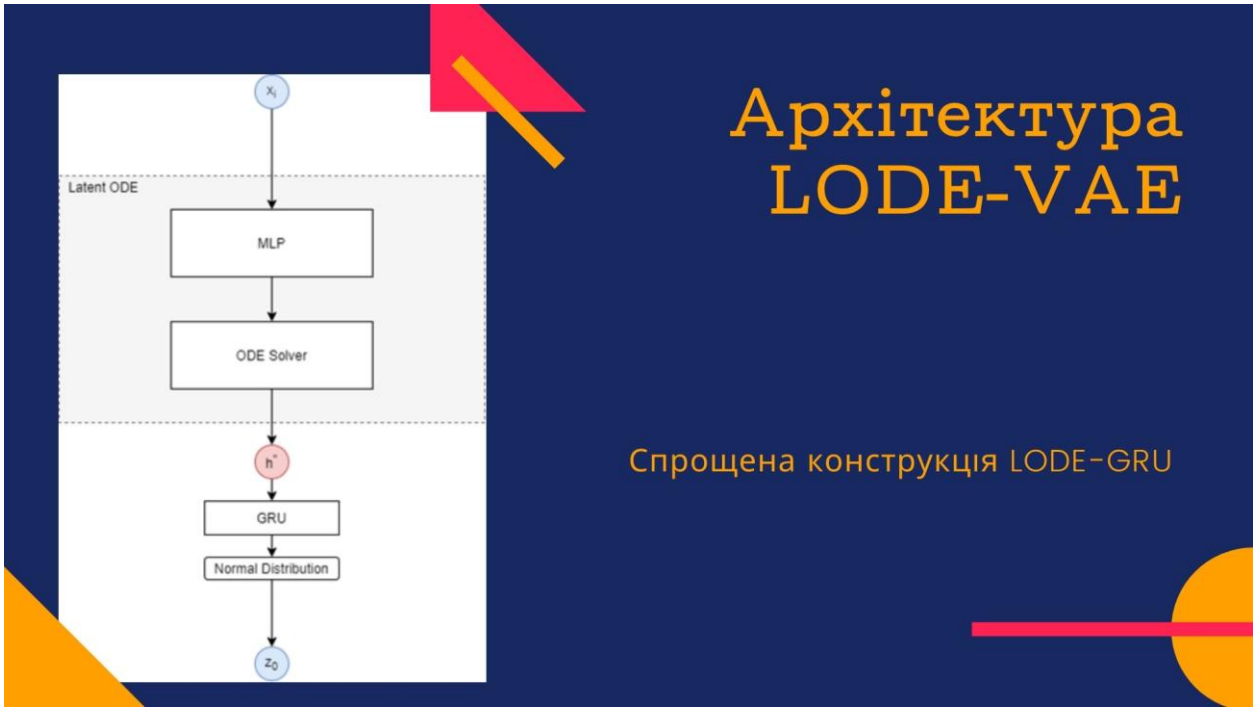
LODE-VAE модель

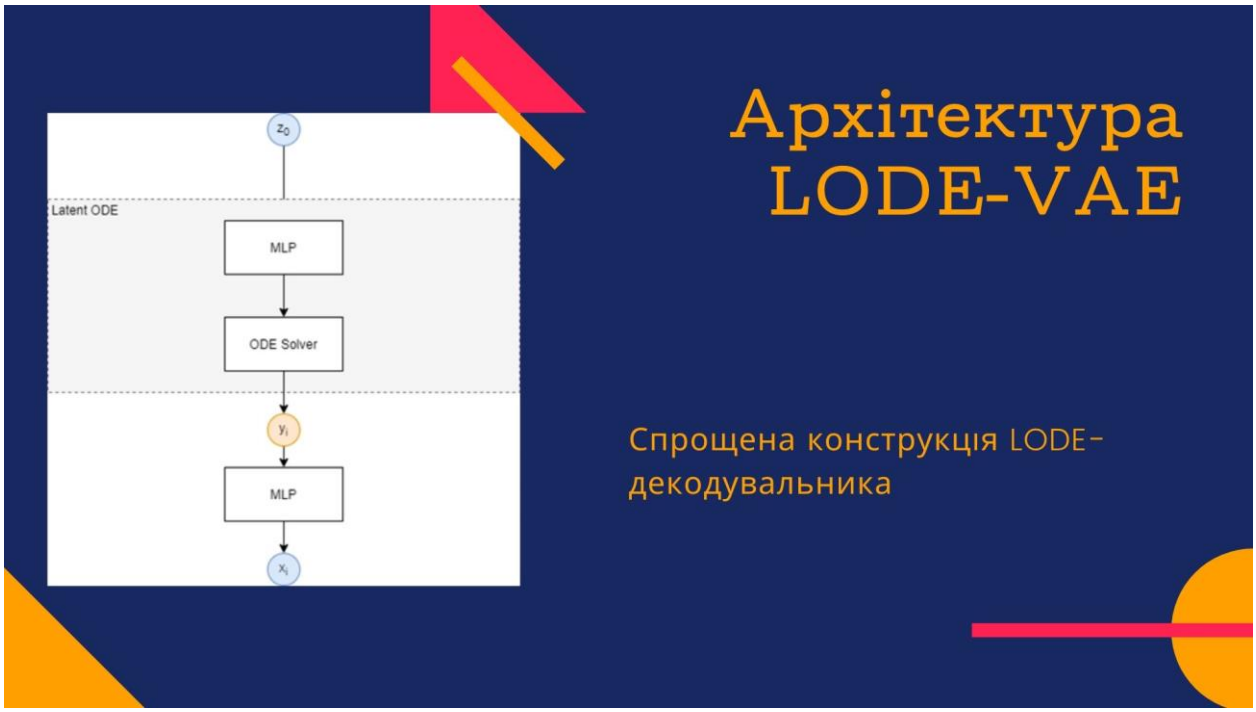
Використання реймворку варіаційних автокодувальників для виявлення прихованих ознак. Моделювання поведінки кожної окремої ознаки LODE-GRU.



Архітектура LODE-VAE

Спрощена конструкція варіаційного автокодувальника


Latent ODE
MLP
ODE Solver
GRU
Normal Distribution
Архітектура LODE-VAE
Спрощена конструкція LODE-GRU


Latent ODE
MLP
ODE Solver
MLP
Архітектура LODE-VAE
Спрощена конструкція LODE-декодувальника

05

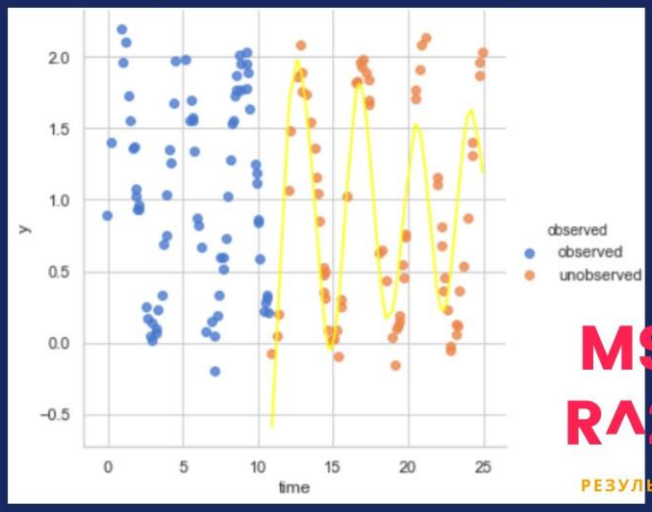
Експеримент та його результати

05

Постановка експерименту

Згенеровано випадкову вибірку з функції x=sin(t).
Випадковим чином обрано 150 точок з інтервалу спостережень.
Необхідно відтворити динаміку ряду з інтервалу спостережень

05



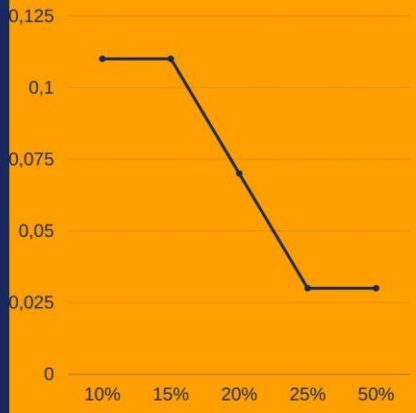
MSE=0.11

R^2=0.88

РЕЗУЛЬТАТ ЕКСПЕРИМЕНТУ

Вплив кількості точок на MSE

Вплив кількості спостережуваних точок (% від загальної кількості) на середньоквадратичну похибку




Висновки
ПРОАНАЛІЗОВАНО
Існуючі підходи до моделювання часових рядів, "класичні" та з використанням машинного навчання
СТВОРЕНО
Модель часового ряду що використовує апарат диференційних рівнянь та використовує парадигми МН
ПРОВЕДЕНО
Експеримент, що доводить працездатність та адекватність моделі.


Дякую за увагу!

ДОДАТОК Б

adams.py

```
import collections

import tensorflow as tf

from solver.misc import (
    _handle_unused_kwargs, _select_initial_step, _convert_to_tensor,
_scaled_dot_product, _is_iterable,
    _optimal_step_size, _compute_error_ratio, move_to_device
)
from solver.solvers import AdaptiveStepsizeODESolver
from solver import compat

_MIN_ORDER = 1
_MAX_ORDER = 12

gamma_star = [
    1, -1 / 2, -1 / 12, -1 / 24, -19 / 720, -3 / 160, -863 / 60480, -275 / 24192, -33953 /
3628800, -0.00789255,
    -0.00678585, -0.00592406, -0.00523669, -0.0046775, -0.00421495, -0.0038269
]

"""Saved state of the variable step size Adams-Bashforth-Moulton solver as
described in
    Solving Ordinary Differential Equations I - Nonstiff Problems III.5
    by Ernst Hairer, Gerhard Wanner, and Syvert P Norsett.
"""
_VCABMState =collections.namedtuple('_VCABMState', 'y_n, prev_f, prev_t,
next_t, phi, order')



def g_and_explicit_phi(prev_t, next_t, implicit_phi, k):
    curr_t = prev_t[0]
    dt = next_t - prev_t[0]

    with tf.device(prev_t[0].device):
        g = tf.Variable(tf.zeros([k + 1]))
```

```
    explicit_phi = collections.deque(maxlen=k)
    beta = move_to_device(tf.convert_to_tensor(1.), prev_t[0].device)

    # tf.assign(g[0], 1)
    compat.assign(g[0], 1)

    c = tf.cast( 1 / move_to_device(tf.range(1, k + 2), prev_t[0].device), tf.float32)
    explicit_phi.append(implicit_phi[0])

    beta = tf.cast(beta, next_t.dtype)
    for j in range(1, k):
        beta = (next_t - prev_t[j - 1]) / (curr_t - prev_t[j]) * beta
        beat_cast = move_to_device(beta, implicit_phi[j][0].device)
        beat_cast = tf.cast(beat_cast, implicit_phi[0][0].dtype)
        explicit_phi.append(tuple(iphi_ * beat_cast for iphi_ in implicit_phi[j]))

        c = c[:-1] - c[1:] if j == 1 else c[:-1] - c[1:] * dt / (next_t - prev_t[j - 1])
        # tf.assign(g[j], tf.cast(c[0], g[j].dtype))
        compat.assign(g[j], tf.cast(c[0], g[j].dtype))
        # g[j] = c[0]

    c = c[:-1] - c[1:] * dt / (next_t - prev_t[k - 1])
    # tf.assign(g[k], tf.cast(c[0], g[k].dtype))
    compat.assign(g[k], tf.cast(c[0], g[k].dtype))

    return g, explicit_phi


def compute_implicit_phi(explicit_phi, f_n, k):
    k = min(len(explicit_phi) + 1, k)
    implicit_phi = collections.deque(maxlen=k)
    implicit_phi.append(f_n)

    def _typesafe_sub(iphi, ephi):
        if ephi.dtype != iphi.dtype:
            ephi = tf.cast(ephi, iphi.dtype)

        return iphi - ephi

    for j in range(1, k):
        implicit_phi.append(
```

```
            tuple(_typesafe_sub(iphi_, ephi_) for iphi_, ephi_ in zip(implicit_phi[j - 1],
explicit_phi[j - 1])))
    return implicit_phi


class VariableCoefficientAdamsBashforth(AdaptiveStepsizeODESolver):

    def __init__(
            self, func, y0, rtol, atol, implicit=True, first_step=None,
max_order=_MAX_ORDER, safety=0.9,
            ifactor=10.0, dfactor=0.2, **unused_kwargs
    ):
        _handle_unused_kwargs(self, unused_kwargs)
        del unused_kwargs

        self.func = func
        self.y0 = y0
        self.rtol = rtol if _is_iterable(rtol) else [rtol] * len(y0)
        self.atol = atol if _is_iterable(atol) else [atol] * len(y0)
        self.implicit = implicit
        self.first_step = first_step
        self.max_order = int(max(_MIN_ORDER, min(max_order,
_MAX_ORDER)))
        self.safety = _convert_to_tensor(safety, dtype=tf.float32,
device=y0[0].device)
        self.ifactor = _convert_to_tensor(ifactor, dtype=tf.float32,
device=y0[0].device)
        self.dfactor = _convert_to_tensor(dfactor, dtype=tf.float32,
device=y0[0].device)

    def before_integrate(self, t):
        prev_f = collections.deque(maxlen=self.max_order + 1)
        prev_t = collections.deque(maxlen=self.max_order + 1)
        phi = collections.deque(maxlen=self.max_order)
        t0 = t[0]
        f0 = self.func(tf.cast(t0, self.y0[0].dtype), self.y0)
        prev_t.appendleft(t0)
        prev_f.appendleft(f0)
        phi.appendleft(f0)
        if self.first_step is None:
```

```
            first_step = _select_initial_step(self.func, t[0], self.y0, 2, self.rtol[0],
self.atol[0], f0=f0)
        else:
            first_step = _select_initial_step(self.func, t[0], self.y0, 2, self.rtol[0],
self.atol[0], f0=f0)
        first_step = move_to_device(first_step, t.device)
        first_step = tf.cast(first_step, t[0].dtype)

        self.vcabm_state = _VCABMState(self.y0, prev_f, prev_t, next_t=t[0] +
first_step, phi=phi, order=1)

    def advance(self, final_t):
        final_t = _convert_to_tensor(final_t,
device=self.vcabm_state.prev_t[0].device)
        while final_t > self.vcabm_state.prev_t[0]:
            # print("VCABM State T = ", final_t.numpy(), self.vcabm_state.y_n)
            self.vcabm_state = self._adaptive_adams_step(self.vcabm_state, final_t)

        assert tf.equal(final_t, self.vcabm_state.prev_t[0])
        return self.vcabm_state.y_n

    def _adaptive_adams_step(self, vcabm_state, final_t):
        y0, prev_f, prev_t, next_t, prev_phi, order = vcabm_state
        if next_t > final_t:
            next_t = final_t
        dt = (next_t - prev_t[0])
        dt_cast = move_to_device(dt, y0[0].device)
        dt_cast = tf.cast(dt_cast, y0[0].dtype)

        # Explicit predictor step.
        g, phi = g_and_explicit_phi(prev_t, next_t, prev_phi, order)
        # g = move_to_device(g, y0[0].device)

        g = tf.cast(g, dt_cast.dtype)
        phi = [tf.cast(phi_, dt_cast.dtype) for phi_ in phi]

        p_next = tuple(
            y0_ + _scaled_dot_product(dt_cast, g[:max(1, order - 1)], phi_[:max(1,
order - 1)])
            for y0_, phi_ in zip(y0, tuple(zip(*phi)))
        )
```

```
        # Update phi to implicit.
        next_t = move_to_device(next_t, p_next[0].device)
        next_f0 = self.func(next_t, p_next)
        implicit_phi_p = compute_implicit_phi(phi, next_f0, order + 1)

        # Implicit corrector step.
        y_next = tuple(
            p_next_ + dt_cast * g[order - 1] * tf.cast(iphi_, dt_cast.dtype)
            for p_next_, iphi_ in zip(p_next, implicit_phi_p[order - 1])
        )

        # Error estimation.
        tolerance = tuple(
            atol_ + rtol_ * tf.reduce_max([tf.abs(y0_), tf.abs(y1_)])
            for atol_, rtol_, y0_, y1_ in zip(self.atol, self.rtol, y0, y_next)
        )
        local_error = tuple(dt_cast * (g[order] - g[order - 1]) * tf.cast(iphi_,
dt_cast.dtype)
                            for iphi_ in implicit_phi_p[order])
        error_k = _compute_error_ratio(local_error, tolerance)
        accept_step = tf.reduce_all((tf.convert_to_tensor(error_k) <= 1))

        if not accept_step:
            # Retry with adjusted step size if step is rejected.
            dt_next = _optimal_step_size(dt, error_k, self.safety, self.ifactor,
self.dfactor, order=order)
            return _VCABMState(y0, prev_f, prev_t, prev_t[0] + dt_next, prev_phi,
order=order)

        # We accept the step. Evaluate f and update phi.
        next_t = move_to_device(next_t, p_next[0].device)
        next_f0 = self.func(next_t, y_next)
        implicit_phi = compute_implicit_phi(phi, next_f0, order + 2)

        next_order = order

        if len(prev_t) <= 4 or order < 3:
            next_order = min(order + 1, 3, self.max_order)
        else:
            implicit_phi_p = [tf.cast(iphi_, dt_cast.dtype) for iphi_ in implicit_phi_p]
```

```
            error_km1 = _compute_error_ratio(
                tuple(dt_cast * (g[order - 1] - g[order - 2]) * iphi_
                      for iphi_ in implicit_phi_p[order - 1]),
                tolerance
            )
            error_km2 = _compute_error_ratio(
                tuple(dt_cast * (g[order - 2] - g[order - 3]) * iphi_
                      for iphi_ in implicit_phi_p[order - 2]),
                tolerance
            )
            if min(error_km1 + error_km2) < max(error_k):
                next_order = order - 1
            elif order < self.max_order:
                error_kp1 = _compute_error_ratio(
                    tuple(dt_cast * gamma_star[order] * iphi_ for iphi_ in
implicit_phi_p[order]), tolerance
                )
                if max(error_kp1) < max(error_k):
                    next_order = order + 1

        # Keep step size constant if increasing order. Else use adaptive step size.
        dt_next = dt if next_order > order else _optimal_step_size(
            dt, error_k, self.safety, self.ifactor, self.dfactor, order=order + 1
        )

        prev_f.appendleft(next_f0)
        prev_t.appendleft(next_t)
        return _VCABMState(p_next, prev_f, prev_t, next_t + dt_next, implicit_phi,
order=next_order)
```

adjoint.py

```
import numpy as np
from typing import Iterable

import tensorflow as tf

from solver.odeint import odeint
from solver.misc import (_flatten, _flatten_convert_none_to_zeros,
                         move_to_device, cast_double, func_cast_double,
                         _check_len, _numel, _convert_to_tensor)
```

```
class _Arguments(object):

    def __init__(self, func, method, options, rtol, atol):
        self.func = func
        self.method = method
        self.options = options
        self.rtol = rtol
        self.atol = atol


_arguments = None


@tf.custom_gradient
def OdeintAdjointMethod(*args):
    global _arguments
    # args = _arguments.args
    # kwargs = _arguments.kwargs
    func = _arguments.func
    method = _arguments.method
    options = _arguments.options
    rtol = _arguments.rtol
    atol = _arguments.atol

    assert len(args) >= 3, 'Internal error: all arguments required.'
    y0, t, flat_params = args[:-2], args[-2], args[-1]

    # registers `t` as a Variable that needs a gred, then resets it to a Tensor
    # for the `odeint` function to work. This is done to force tf to allow us to
    # pass the gradient of t as output.
    # t = tf.get_variable('t', initializer=t)
    # t = tf.convert_to_tensor(t, dtype=t.dtype)

    ans = odeint(func, y0, t, rtol=rtol, atol=atol, method=method, options=options)

    @func_cast_double
    def grad(*grad_output, variables=None):
        global _arguments
        # t, flat_params, *ans = ctx.saved_tensors
```

```
        # ans = tuple(ans)
        # func, rtol, atol, method, options = ctx.func, ctx.rtol, ctx.atol, ctx.method,
ctx.options
        func = _arguments.func
        method = _arguments.method
        options = _arguments.options
        rtol = _arguments.rtol
        atol = _arguments.atol

        print("Gradient Output : ", grad_output)
        print("Variables : ", variables)

        n_tensors = len(ans)
        f_params = tuple(variables)

        # TODO: use a keras.Model and call odeint_adjoint to implement higher
order derivatives.
        def augmented_dynamics(t, y_aug):
            # Dynamics of the original system augmented with
            # the adjoint wrt y, and an integrator wrt t and args.
            # get y and adj_y
            y, _ = y_aug[:n_tensors], y_aug[n_tensors:2 * n_tensors]  # Ignore
adj_time and adj_params.

            # t = tf.get_variable('t', initializer=t)
            # y = tuple(tf.Variable(y_) for y_ in y)

            with tf.GradientTape() as tape:
                tape.watch(t)
                tape.watch(y)
                func_eval = func(t, y)
                func_eval = cast_double(func_eval)

            # print('y', [y_.numpy().shape for y_ in y])
            # print('adj y', [a.numpy().shape for a in adj_y])

            vjp_t, *vjp_y_and_params = tape.gradient(func_eval, (t,) + y + f_params,
                                                     # list(-adj_y_ for adj_y_ in adj_y),
                                                     )

            vjp_y = vjp_y_and_params[:n_tensors]
```

```
    vjp_params = vjp_y_and_params[n_tensors:]
    # print('vjp_y', [v.numpy().shape if v is not None else None for v in vjp_y])
    # print()

    # autograd.grad returns None if no gradient, set to zero.
    vjp_t = tf.zeros_like(t, dtype=t.dtype) if vjp_t is None else vjp_t
    vjp_y = tuple(tf.zeros_like(y_, dtype=y_.dtype)
                  if vjp_y_ is None else vjp_y_
                  for vjp_y_, y_ in zip(vjp_y, y))
    vjp_params = _flatten_convert_none_to_zeros(vjp_params, f_params)

    if _check_len(f_params) == 0:
        vjp_params = tf.convert_to_tensor(0., dtype=vjp_y[0].dype)
        vjp_params = move_to_device(vjp_params, vjp_y[0].device)

    # print('vjp_t grad', vjp_t.numpy())
    # print('vjp_params', [v.numpy() for v in vjp_params])
    # print('vjp y grads', [v.numpy().shape for v in vjp_y])
    # print("LEN FUNC EVALS : ", len(func_eval))
    # print()

    return (*func_eval, *vjp_y, vjp_t, vjp_params)

T = ans[0].shape[0]
if isinstance(grad_output, tf.Tensor) or isinstance(grad_output, tf.Variable):
    adj_y = [grad_output[-1]]
else:
    adj_y = tuple(grad_output_[-1] for grad_output_ in grad_output)
# adj_y = tuple(grad_output_[-1] for grad_output_ in grad_output)
adj_params = tf.zeros_like(flat_params, dtype=flat_params.dtype)
adj_time = move_to_device(tf.convert_to_tensor(0., dtype=t.dtype), t.device)
time_vjps = []
for i in range(T - 1, 0, -1):

    ans_i = tuple(ans_[i] for ans_ in ans)

    if isinstance(grad_output, tf.Tensor) or isinstance(grad_output, tf.Variable):
        grad_output_i = [grad_output[i]]
    else:
        grad_output_i = tuple(grad_output_[i] for grad_output_ in grad_output)
```

```
            func_i = func(t[i], ans_i)
            func_i = cast_double(func_i)

            if not isinstance(func_i, Iterable):
                func_i = [func_i]

            # Compute the effect of moving the current time measurement point.
            dLd_cur_t = sum(
                tf.reshape(tf.matmul(tf.reshape(func_i_, [1, -1]),
tf.reshape(grad_output_i_, [-1, 1])), [1])
                for func_i_, grad_output_i_ in zip(func_i, grad_output_i)
            )
            adj_time = cast_double(adj_time)
            adj_time = adj_time - dLd_cur_t
            time_vjps.append(dLd_cur_t)

            # Run the augmented system backwards in time.
            if isinstance(adj_params, Iterable):
                count = _numel(adj_params)

                if count == 0:
                    adj_params = move_to_device(tf.convert_to_tensor(0.,
dtype=adj_y[0].dtype), adj_y[0].device)

            aug_y0 = (*ans_i, *adj_y, adj_time, adj_params)

            # print('ans i', [a.numpy().shape for a in ans_i])
            # print('adj y', [a.numpy().shape for a in adj_y])
            # print('adj time', adj_time.numpy().shape)
            # print('adj params', adj_params.numpy().shape)

            # print()

            aug_ans = odeint(
                augmented_dynamics,
                aug_y0,
                tf.convert_to_tensor([t[i], t[i - 1]]),
                rtol=rtol, atol=atol, method=method, options=options
            )

            # Unpack aug_ans.
```

```
    adj_y = aug_ans[n_tensors:2 * n_tensors]
    adj_time = aug_ans[2 * n_tensors]
    adj_params = aug_ans[2 * n_tensors + 1]

    adj_y = tuple(adj_y_[1] if _check_len(adj_y_) > 0
                  else adj_y_ for adj_y_ in adj_y)
    if _check_len(adj_time) > 0:
        adj_time = adj_time[1]
    if _check_len(adj_params) > 0:
        adj_params = adj_params[1]

    adj_y = tuple(adj_y_ + grad_output_[i - 1] for adj_y_,
                  grad_output_ in zip(adj_y, grad_output))

    del aug_y0, aug_ans

time_vjps.append(adj_time)
time_vjps = tf.concat(time_vjps[::-1], 0)

print()
print('adj y', len(adj_y))
print('time vjps', time_vjps.shape)
print('adj params', adj_params.shape)
print()

# reshape the parameters back into the correct variable shapes
var_flat_lens = [_numel(v, dtype=tf.int32).numpy() for v in variables]
var_shapes = [v.shape for v in variables]

adj_params_splits = tf.split(adj_params, var_flat_lens)
adj_params_list = [tf.reshape(p, v_shape)
                   for p, v_shape in zip(adj_params_splits, var_shapes)]

# add the time gradient (always the first tensor in list of variables)
# adj_params.insert(0, time_vjps)

# adj_y_grad_vars = list(zip(adj_y, grad_output))
# time_grad_vars = list((time_vjps, t))
model_vars = list(adj_params_list)  # list(zip(adj_params, variables))

grad_vars = model_vars  # adj_y_grad_vars + time_grad_vars + model_vars
```

```
        # print('adj y grad', len(adj_y_grad_vars))
        # print('time grad', len(time_grad_vars))
        print('model grad', len(model_vars))
        print('model grad values', [v for v in grad_vars])
        print()

        # if len(adj_y) == 1:
        #     adj_y = adj_y[0]

        return (adj_y, model_vars)
        # print("Grad_vars : ", grad_vars)
        # return zip(*grad_vars)  # (*adj_y, time_vjps, adj_params, None, None)
        # return (*adj_y, time_vjps, adj_params)

    return ans, grad


def odeint_adjoint(func, y0, t, rtol=1e-6, atol=1e-12, method=None,
options=None):
    """
        Adjoint Equation Method for solving ODE
    """
    # We need this in order to access the variables inside this module,
    # since we have no other way of getting variables along the execution path.
    if not isinstance(func, tf.keras.models.Model):
        raise ValueError('func is required to be an instance of Model')

    with tf.python.eager.context.eager_mode():
        tensor_input = False
        if tf.debugging.is_numeric_tensor(y0):
            class TupleFunc(tf.keras.models.Model):

                def __init__(self, base_func, **kwargs):
                    super(TupleFunc, self).__init__(**kwargs)
                    self.base_func = base_func

                def call(self, t, y):
                    return (self.base_func(t, y[0]),)

            tensor_input = True
            y0 = (y0,)
```

```
        func = TupleFunc(func)

    # build the function to get its variables
    if not func.built:
        _ = func(t, y0)

    flat_params = _flatten(func.variables)

    global _arguments
    _arguments = _Arguments(func, method, options, rtol, atol)

    ys = OdeintAdjointMethod(*y0, t, flat_params)

    if tensor_input or type(ys) == tuple or type(ys) == list:
        ys = ys[0]

    return ys
```

dopri5.py

```
# Based on
https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/integrate
import tensorflow as tf
from solver.misc import (
    _scaled_dot_product, _convert_to_tensor, _is_finite, _select_initial_step,
_handle_unused_kwargs, _is_iterable,
    _optimal_step_size, _compute_error_ratio, move_to_device, cast_double
)
from solver.solvers import AdaptiveStepsizeODESolver
from solver.interp import _interp_fit, _interp_evaluate
from solver.rk_common import _RungeKuttaState, _ButcherTableau,
_runge_kutta_step

_DORMAND_PRINCE_SHAMPINE_TABLEAU = _ButcherTableau(
    alpha=[1 / 5, 3 / 10, 4 / 5, 8 / 9, 1., 1.],
    beta=[
        [1 / 5],
        [3 / 40, 9 / 40],
        [44 / 45, -56 / 15, 32 / 9],
        [19372 / 6561, -25360 / 2187, 64448 / 6561, -212 / 729],
        [9017 / 3168, -355 / 33, 46732 / 5247, 49 / 176, -5103 / 18656],
        [35 / 384, 0, 500 / 1113, 125 / 192, -2187 / 6784, 11 / 84],
```

```
    ],
    c_sol=[35 / 384, 0, 500 / 1113, 125 / 192, -2187 / 6784, 11 / 84, 0],
    c_error=[
        35 / 384 - 1951 / 21600,
        0,
        500 / 1113 - 22642 / 50085,
        125 / 192 - 451 / 720,
        -2187 / 6784 - -12231 / 42400,
        11 / 84 - 649 / 6300,
        -1. / 60.,
    ],
)

DPS_C_MID = [
    6025192743 / 30085553152 / 2, 0, 51252292925 / 65400821598 / 2, -
2691868925 / 45128329728 / 2,
    187940372067 / 1594534317056 / 2, -1776094331 / 19743644256 / 2,
11237099 / 235043384 / 2
]


def _interp_fit_dopri5(y0, y1, k, dt,
tableau=_DORMAND_PRINCE_SHAMPINE_TABLEAU):
    """Fit an interpolating polynomial to the results of a Runge-Kutta step."""
    dt = cast_double(dt)
    y0 = cast_double(y0)

    y_mid = tuple(y0_ + _scaled_dot_product(dt, DPS_C_MID, k_) for y0_, k_ in
zip(y0, k))
    f0 = tuple(k_[0] for k_ in k)
    f1 = tuple(k_[-1] for k_ in k)
    return _interp_fit(y0, y1, y_mid, f0, f1, dt)


def _abs_square(x):
    return x * x


def _ta_append(list_of_tensors, value):
    """Append a value to the end of a list of PyTorch tensors."""
    list_of_tensors.append(value)
```

```
    return list_of_tensors


class Dopri5Solver(AdaptiveStepsizeODESolver):

    def __init__(
        self, func, y0, rtol, atol, first_step=None, safety=0.9, ifactor=10.0,
dfactor=0.2, max_num_steps=2**31 - 1,
        tableau=None, **unused_kwargs
    ):
        _handle_unused_kwargs(self, unused_kwargs)
        del unused_kwargs

        self.func = func
        self.y0 = y0
        self.rtol = rtol if _is_iterable(rtol) else [rtol] * len(y0)
        self.atol = atol if _is_iterable(atol) else [atol] * len(y0)
        self.first_step = first_step
        self.safety = _convert_to_tensor(safety, dtype=tf.float32,
device=y0[0].device)
        self.ifactor = _convert_to_tensor(ifactor, dtype=tf.float32,
device=y0[0].device)
        self.dfactor = _convert_to_tensor(dfactor, dtype=tf.float32,
device=y0[0].device)
        self.max_num_steps = _convert_to_tensor(max_num_steps, dtype=tf.int32,
device=y0[0].device)
        self.tableau = tableau if tableau is not None else
_DORMAND_PRINCE_SHAMPINE_TABLEAU
        self.order = 5

    def before_integrate(self, t):
        f0 = self.func(tf.cast(t[0], self.y0[0].dtype),self.y0)
        if self.first_step is None:
            first_step = move_to_device(_select_initial_step(self.func, t[0], self.y0, 4,
self.rtol[0], self.atol[0], f0=f0), t.device)
        else:
            first_step = _convert_to_tensor(0.01, dtype=t.dtype, device=t.device)
        self.rk_state = _RungeKuttaState(self.y0, f0, t[0], t[0], first_step,
interp_coeff=[self.y0] * 5)

    def advance(self, next_t):
```

```
      """Interpolate through the next time point, integrating as necessary."""
      n_steps = 0
      while next_t > self.rk_state.t1:
        assert n_steps < self.max_num_steps, 'max_num_steps exceeded
({}>={})'.format(n_steps, self.max_num_steps)
        self.rk_state = self._adaptive_dopri5_step(self.rk_state)
        n_steps += 1
      return _interp_evaluate(self.rk_state.interp_coeff, self.rk_state.t0,
self.rk_state.t1, next_t)

    def _adaptive_dopri5_step(self, rk_state):
      """Take an adaptive Runge-Kutta step to integrate the ODE."""
      y0, f0, _, t0, dt, interp_coeff = rk_state
      ########################################################
      #                    Assertions                        #
      ########################################################
      dt = tf.cast(dt, t0.dtype)
      assert t0 + dt > t0, 'underflow in dt {}'.format(dt.numpy())
      for y0_ in y0:
        assert _is_finite(tf.abs(y0_)), 'non-finite values in state `y`: {}'.format(y0_)
      y1, f1, y1_error, k = _runge_kutta_step(self.func, y0, f0, t0, dt,
tableau=self.tableau)

      ########################################################
      #                    Error Ratio                       #
      ########################################################
      mean_sq_error_ratio = _compute_error_ratio(y1_error, atol=self.atol,
rtol=self.rtol, y0=y0, y1=y1)
      accept_step = tf.reduce_all(tf.convert_to_tensor(mean_sq_error_ratio) <= 1)

      ########################################################
      #                    Update RK State                   #
      ########################################################
      y_next = y1 if accept_step else y0
      f_next = f1 if accept_step else f0
      t_next = t0 + dt if accept_step else t0
      interp_coeff = _interp_fit_dopri5(y0, y1, k, dt) if accept_step else
interp_coeff
      dt_next = _optimal_step_size(
        dt, mean_sq_error_ratio, safety=self.safety, ifactor=self.ifactor,
dfactor=self.dfactor, order=self.order
```

```
        )
        rk_state = _RungeKuttaState(y_next, f_next, t0, t_next, dt_next, interp_coeff)
        return rk_state
```

fixed_adams.py

```
import collections
import sys

from solver import rk_common
from solver.misc import _scaled_dot_product, _has_converged, func_cast_double
from solver.solvers import FixedGridODESolver

_BASHFORTH_COEFFICIENTS = [
    [],  # order 0
    [11],
    [3, -1],
    [23, -16, 5],
    [55, -59, 37, -9],
    [1901, -2774, 2616, -1274, 251],
    [4277, -7923, 9982, -7298, 2877, -475],
    [198721, -447288, 705549, -688256, 407139, -134472, 19087],
    [434241, -1152169, 2183877, -2664477, 2102243, -1041723, 295767, -36799],
    [14097247, -43125206, 95476786, -139855262, 137968480, -91172642,
38833486, -9664106, 1070017],
    [30277247, -104995189, 265932680, -454661776, 538363838, -444772162,
252618224, -94307320, 20884811, -2082753],
    [
        2132509567, -8271795124, 23591063805, -46113029016, 63716378958, -
63176201472, 44857168434, -22329634920,
        7417904451, -1479574348, 134211265
    ],
    [
        4527766399, -19433810163, 61633227185, -135579356757, 214139355366,
-247741639374, 211103573298, -131365867290,
        58189107627, -17410248271, 3158642445, -262747265
    ],
    [
        13064406523627, -61497552797274, 214696591002612, -
524924579905150, 932884546055895, -1233589244941764,
        1226443086129408, -915883387152444, 507140369728425, -
202322913738370, 55060974662412, -9160551085734,
```

703604254357
],
[
27511554976875, -140970750679621, 537247052515662, -1445313351681906, 2854429571790805, -4246767353305755,
4825671323488452, -4204551925534524, 2793869602879077, -1393306307155755, 505586141196430, -126174972681906,
19382853593787, -1382741929621
],
[
173233498598849, -960122866404112, 3966421670215481, -11643637530577472, 25298910337081429, -41825269932507728,
53471026659940509, -53246738660646912, 41280216336284259, -24704503655607728, 11205849753515179,
-3728807256577472, 859236476684231, -122594813904112, 8164168737599
],
[
362555126427073, -2161567671248849, 9622096909515337, -30607373860520569, 72558117072259733,
-131963191940828581, 187463140112902893, -210020588912321949, 186087544263596643, -129930094104237331,
70724351582843483, -29417910911251819, 9038571752734087, -1934443196892599, 257650275915823, -16088129229375
],
[
192996103681340479, -12318873395934449 74, 5878428128276811750, -20141834622844109630, 51733880057282977010,
-102651404730855807942, 160414858999474733422, -199694296833704562550, 199061418623907202560,
-158848144481581407370, 100878076849144434322, -50353311405771659322, 19338911944324897550,
-5518639984393844930, 110256034514105961 0, -137692773163513234, 8092989203533249
],
[
401972381695456831, -27354376428440797 89, 139301599658111422 28, -51150187791975812900, 141500575026572531760,
-304188128232928718008, 518600355541383671092, -710171024091234303204, 786600875277595877750,

```
        -706174326992944287370, 512538584122114046748, -
29847726035397752289 2, 137563142659866897224,
        -49070094880794267600, 13071639236569712860, -
24486892555845451 96, 287848942064256339, -15980174332775873
    ],
    [
        333374427829017307697, -2409687649238345289684,
130441391399831833251471, -51099831122607588046344,
        151474888613495715415020, -35070292960829145516 7896,
64775815749192190229 2692, -96771374654462965869 0408,
        11790787437862804519 53222, -11761618299567683652 19840,
96037703544420595081 3626, -63918212308229874800 1432,
        34369046161247151674 6028, -14711873899328816374 2312,
48988597853073465932 820, -12236035290567356418 552,
        21575749428818183120 49, -239560589366324764716,
12600467236042756559
    ],
    [
        691668239157222107697, -5292843584961252933125,
30349492858024727686755, -12634654485592785613 4295,
        39953730766984215099 6468, -99116845054513507083 5076,
19716290280837988457 50380, -31910653888463186795 44380,
        42416143312081499471 51790, -46543264688014788944 06214,
42227568797763540655 93786, -31618210899001865392 48210,
        19430188189820023956 55620, -97035019108653136864 9620,
38773978703469909236 4924, -12105960102398543300 3532,
        28462032496476316665 705, -474033575709371071 3245,
49866922095664786687 5, -24919383499187492303
    ],
]

_MOULTON_COEFFICIENTS = [
    [],  # order 0
    [1],
    [1, 1],
    [5, 8, -1],
    [9, 19, -5, 1],
    [251, 646, -264, 106, -19],
    [475, 1427, -798, 482, -173, 27],
    [19087, 65112, -46461, 37504, -20211, 6312, -863],
    [36799, 139849, -121797, 123133, -88547, 41499, -11351, 1375],
```

[1070017, 4467094, -4604594, 5595358, -5033120, 3146338, -1291214, 312874, -33953],
[2082753, 9449717, -11271304, 16002320, -17283646, 13510082, -7394032, 2687864, -583435, 57281],
[
134211265, 656185652, -890175549, 1446205080, -1823311566, 1710774528, -1170597042, 567450984, -184776195,
36284876, -3250433
],
[
262747265, 1374799219, -2092490673, 3828828885, -5519460582, 6043521486, -4963166514, 3007739418, -1305971115,
384709327, -68928781, 5675265
],
[
703604254357, 3917551216986, -6616420957428, 13465774256510, -21847538039895, 27345870698436, -26204344465152,
19058185652796, -10344711794985, 4063327863170, -1092096992268, 179842822566, -13695779093
],
[
1382741929621, 8153167962181, -15141235084110, 33928990133618, -61188680131285, 86180228689563, -94393338653892,
80101021029180, -52177910882661, 25620259777835, -9181635605134, 2268078814386, -345457086395, 24466579093
],
[
8164168737599, 50770967534864, -102885148956217, 251724894607936, -499547203754837, 781911618071632,
-963605400824733, 934600833490944, -710312834197347, 418551804601264, -187504936597931, 61759426692544,
-14110480969927, 1998759236336, -132282840127
],
[
16088129229375, 105145058757073, -230992163723849, 612744541065337, -1326978663058069, 2285168598349733,
-3129453071993581, 3414941728852893, -2966365730265699, 2039345879546643, -1096355235402331, 451403108933483,
-137515713789319, 29219384284087, -3867689367599, 240208245823
],
[

```
        8092989203533249, 55415287221275246, -131240807912923110,
375195469874202430, -880520318434977010,
        1654462865819232198, -2492570347928318318, 3022404969160106870, -
2953729295811279360, 2320851086013919370,
        -1455690451266780818, 719242466216944698, -273894214307914510,
77597639915764930, -15407325991235610,
        1913813460537746, -111956703448001
    ],
    [
        15980174332775873, 114329243705491117, -290470969929371220,
890337710266029860, -2250854333681641520,
        4582441343348851896, -7532171919277411636, 10047287575124288740, -
10910555637627652470, 9644799218032932490,
        -6913858539337636636, 3985516155854664396, -1821304040326216520,
645008976643217360, -170761422500096220,
        31816981024600492, -3722582669836627, 205804074290625
    ],
    [
        12600467236042756559, 93965550344204933076, -
25500775187503391809 5, 834286388106402145800,
        -2260420115705863623660, 495665559279054214 6968, -
882705255997938420 9108, 12845814402199484797800,
        -15345231910046032448070, 150727814551226865459 20, -
12155867625610599812538, 800852080962232457 1288,
        -4269779992576330506540, 181458456415944578724 0, -
600505972582990474260, 149186846171741510136,
        -2618253884192531288 1, 2895045518506940460, -151711881512390095
    ],
    [
        24919383499187492303, 193280569173472261637, -
55816072011562939555 5, 194139566895098646133 5,
        -561213180236445592626 0, 131871858984392703307 56, -
25293146116627869170796, 398784192267844424218 20,
        -5197064945367027413547 0, 561546786846187399399 10, -
50320851025594566473146, 372972272528228583819 06,
        -2272635040753813383930 0, 112682101249879923270 60, -
447488665802416698534 0, 138966526329621169921 2,
        -325187970422032795497, 53935307402575440285, -
5652892248087175675, 281550972898020815
    ],
]
```

```
_DIVISOR = [
    None, 11, 2, 12, 24, 720, 1440, 60480, 120960, 3628800, 7257600, 479001600,
958003200, 2615348736000, 5230697472000,
    31384184832000, 62768369664000, 32011868528640000,
64023737057280000, 51090942171709440000, 102181884343418880000
]

_MIN_ORDER = 4
_MAX_ORDER = 12
_MAX_ITERS = 4


class AdamsBashforthMoulton(FixedGridODESolver):

    def __init__(
            self, func, y0, rtol=1e-3, atol=1e-4, implicit=True,
max_iters=_MAX_ITERS, max_order=_MAX_ORDER, **kwargs
    ):
        super(AdamsBashforthMoulton, self).__init__(func, y0, **kwargs)

        self.rtol = rtol
        self.atol = atol
        self.implicit = implicit
        self.max_iters = max_iters
        self.max_order = int(min(max_order, _MAX_ORDER))
        self.prev_f = collections.deque(maxlen=self.max_order - 1)
        self.prev_t = None

    def _update_history(self, t, f):
        if self.prev_t is None or self.prev_t != t:
            self.prev_f.appendleft(f)
            self.prev_t = t

    @func_cast_double
    def step_func(self, func, t, dt, y):
        self._update_history(t, func(t, y))
        order = min(len(self.prev_f), self.max_order - 1)
        if order < _MIN_ORDER - 1:
            # Compute using RK4.
            dy = rk_common.rk4_alt_step_func(func, t, dt, y, k1=self.prev_f[0])
```

```
            return dy
        else:
            # Adams-Bashforth predictor.
            bashforth_coeffs = _BASHFORTH_COEFFICIENTS[order]
            ab_div = _DIVISOR[order]
            dy = tuple(dt * _scaled_dot_product(1 / ab_div, bashforth_coeffs, f_) for f_
in zip(*self.prev_f))

            # Adams-Moulton corrector.
            if self.implicit:
                moulton_coeffs = _MOULTON_COEFFICIENTS[order + 1]
                am_div = _DIVISOR[order + 1]
                delta = tuple(dt * _scaled_dot_product(1 / am_div, moulton_coeffs[1:],
f_) for f_ in zip(*self.prev_f))
                converged = False
                for _ in range(self.max_iters):
                    dy_old = dy
                    f = func(t + dt, tuple(y_ + dy_ for y_, dy_ in zip(y, dy)))
                    dy = tuple(dt * (moulton_coeffs[0] / am_div) * f_ + delta_ for f_,
delta_ in zip(f, delta))
                    converged = _has_converged(dy_old, dy, self.rtol, self.atol)
                    if converged:
                        break
                if not converged:
                    print('Warning: Functional iteration did not converge. Solution may be
incorrect.', file=sys.stderr)
                    self.prev_f.pop()
                self._update_history(t, f)
            return dy

    @property
    def order(self):
        return 4


class AdamsBashforth(AdamsBashforthMoulton):

    def __init__(self, func, y0, **kwargs):
        super(AdamsBashforth, self).__init__(func, y0, implicit=False, **kwargs)
```

fixed_grid.py

```
from solver import rk_common
from solver.misc import func_cast_double, cast_double
from solver.solvers import FixedGridODESolver


class Euler(FixedGridODESolver):

    @func_cast_double
    def step_func(self, func, t, dt, y):
        return tuple(dt * f_ for f_ in cast_double(func(t, y)))

    @property
    def order(self):
        return 1


class Midpoint(FixedGridODESolver):

    @func_cast_double
    def step_func(self, func, t, dt, y):
        y_mid = tuple(y_ + f_ * dt / 2 for y_, f_ in zip(y, cast_double(func(t, y))))
        return tuple(dt * f_ for f_ in cast_double(func(t + dt / 2, y_mid)))

    @property
    def order(self):
        return 2


class Huen(FixedGridODESolver):

    @func_cast_double
    def step_func(self, func, t, dt, y):
        f_outs = cast_double(func(t, y))
        ft_1_hat = tuple(y_ + dt * f_ for y_, f_ in zip(y, f_outs))
        ft_1_outs = cast_double(func(t + dt, ft_1_hat))
        return tuple(dt / 2. * (ft_ + ft_1_hat_) for ft_, ft_1_hat_ in zip(f_outs,
                ft_1_outs))

    @property
    def order(self):
        return 2
```

```
class RK4(FixedGridODESolver):

    @func_cast_double
    def step_func(self, func, t, dt, y):
        return rk_common.rk4_alt_step_func(func, t, dt, y)

    @property
    def order(self):
        return 4
```

interp.py

```
import tensorflow as tf

from solver.misc import _convert_to_tensor, _dot_product, move_to_device


def _interp_fit(y0, y1, y_mid, f0, f1, dt):
    """Fit coefficients for 4th order polynomial interpolation.
    Args:
        y0: function value at the start of the interval.
        y1: function value at the end of the interval.
        y_mid: function value at the mid-point of the interval.
        f0: derivative value at the start of the interval.
        f1: derivative value at the end of the interval.
        dt: width of the interval.
    Returns:
        List of coefficients `[a, b, c, d, e]` for interpolating with the polynomial
        `p = a * x ** 4 + b * x ** 3 + c * x ** 2 + d * x + e` for values of `x`
        between 0 (start of interval) and 1 (end of interval).
    """
    a = tuple(
        _dot_product([-2 * dt, 2 * dt, -8, -8, 16], [f0_, f1_, y0_, y1_, y_mid_])
        for f0_, f1_, y0_, y1_, y_mid_ in zip(f0, f1, y0, y1, y_mid)
    )
    b = tuple(
        _dot_product([5 * dt, -3 * dt, 18, 14, -32], [f0_, f1_, y0_, y1_, y_mid_])
        for f0_, f1_, y0_, y1_, y_mid_ in zip(f0, f1, y0, y1, y_mid)
    )
    c = tuple(
```

```
        _dot_product([-4 * dt, dt, -11, -5, 16], [f0_, f1_, y0_, y1_, y_mid_])
        for f0_, f1_, y0_, y1_, y_mid_ in zip(f0, f1, y0, y1, y_mid)
    )
    d = tuple(dt * f0_ for f0_ in f0)
    e = y0
    return [a, b, c, d, e]


def _interp_evaluate(coefficients, t0, t1, t):
    """Evaluate polynomial interpolation at the given time point.
    Args:
        coefficients: list of Tensor coefficients as created by `interp_fit`.
        t0: scalar float64 Tensor giving the start of the interval.
        t1: scalar float64 Tensor giving the end of the interval.
        t: scalar float64 Tensor giving the desired interpolation point.
    Returns:
        Polynomial interpolation of the coefficients at time `t`.
    """

    dtype = coefficients[0][0].dtype
    device = coefficients[0][0].device

    t0 = _convert_to_tensor(t0, dtype=dtype, device=device)
    t1 = _convert_to_tensor(t1, dtype=dtype, device=device)
    t = _convert_to_tensor(t, dtype=dtype, device=device)

    assert (t0 <= t) & (t <= t1), 'invalid interpolation, fails `t0 <= t <= t1`: {}, {},
                {}'.format(t0, t, t1)
    x = tf.cast(((t - t0) / (t1 - t0)), dtype)
    x = move_to_device(x, device)

    xs = [move_to_device(tf.convert_to_tensor(1, dtype=dtype), device), x]
    for _ in range(2, len(coefficients)):
        xs.append(xs[-1] * x)

    return tuple(_dot_product(coefficients_, reversed(xs)) for coefficients_
            in zip(*coefficients))
```

losses.py

```
import numpy as np
import tensorflow as tf
```

```
import tensorflow_probability as tfp

def gaussian_log_likelihood(mu_2d, data_2d, obsrv_std, indices = None):
    n_data_points = mu_2d.shape[-1]
    if n_data_points > 0:
        gaussian = tfp.distributions.Independent(tfp.distributions.Normal(loc =
                mu_2d,
                                        scale =
                tf.tile(tf.expand_dims(obsrv_std,0),[n_data_points])), 1)
        log_prob = gaussian.log_prob(data_2d)
        log_prob = log_prob / n_data_points
    else:
        log_prob = tf.squeeze(tf.zeros([1]))
    return log_prob

def poisson_log_likelihood(log_lambdas, data, indices, int_lambdas):
    n_data_points = data.shape[-1]

    if n_data_points > 0:
        log_prob = tf.reduce_sum(log_lambdas) - int_lambdas[indices]
    else:
        log_prob = tf.squeeze(tf.zeros([1]))
    return log_prob

def mse_wrapper(mu, data, indices=None):
    n_data_points = data.shape[-1]

    if n_data_points > 0:
        loss = tf.keras.losses.MeanSquaredError()(data, mu)
    else:
        loss = tf.squeeze(tf.zeros([1]))
    return loss

def gaussian_log_density(mu, data, obsrv_std):
    if (len(mu.shape) == 3):
        # add additional dimension for gp samples
        mu = tf.expand_dims(mu, 0)

    if (len(data.shape) == 2):
        # add additional dimension for gp samples and time step
        data = tf.expand_dims(tf.expand_dims(data, 0), 2)
```

```
    elif (len(data.shape) == 3):
        # add additional dimension for gp samples
        data = tf.expand_dims(data, 0)

    n_traj_samples, n_traj, n_timepoints, n_dims = mu.shape

    assert(data.shape[-1] == n_dims)

    # Shape after permutation: [n_traj, n_traj_samples, n_timepoints, n_dims]
    mu_flat =tf.reshape(mu, [n_traj_samples*n_traj, n_timepoints * n_dims])
    n_traj_samples, n_traj, n_timepoints, n_dims = data.shape
    data_flat = tf.reshape(data, [n_traj_samples*n_traj, n_timepoints * n_dims])
    res = gaussian_log_likelihood(mu_flat, data_flat, tf.cast(obsrv_std, tf.float32))
    res = tf.transpose(tf.reshape(res, [n_traj_samples, n_traj]), perm=[0,1])
    return res

def mse(mu, data):
    if (len(mu.shape) == 3):
        # add additional dimension for gp samples
        mu = tf.expand_dims(mu, 0)

    if (len(data.shape) == 2):
        # add additional dimension for gp samples and time step
        data = tf.expand_dims(tf.expand_dims(data, 0), 2)
    elif (len(data.shape) == 3):
        # add additional dimension for gp samples
        data = tf.expand_dims(data, 0)

    n_traj_samples, n_traj, n_timepoints, n_dims = mu.shape

    assert(data.shape[-1] == n_dims)

    # Shape after permutation: [n_traj, n_traj_samples, n_timepoints, n_dims]
    mu_flat =tf.reshape(mu, [n_traj_samples*n_traj, n_timepoints * n_dims])
    n_traj_samples, n_traj, n_timepoints, n_dims = data.shape
    data_flat = tf.reshape(data, [n_traj_samples*n_traj, n_timepoints * n_dims])

    res = mse_wrapper(mu_flat, data_flat)
    res = tf.transpose(tf.reshape(res, [n_traj_samples, n_traj]), perm=[0,1])
    return res
```

```
def poisson_proc_likelihood(truth, pred_y, info):
    # Compute Poisson likelihood
    # Sum log lambdas across all time points

    poisson_log_l = tf.reduce_sum(info["log_lambda_y"], 2) - info["int_lambda"]
    # Sum over data dims
    poisson_log_l = tf.reduce_mean(poisson_log_l, -1)

    return poisson_log_l

def binary_ce(pred_y, truth):

    truth = tf.reshape(truth, (-1,))

    if len(pred_y.shape) == 1:
        pred_y = tf.expand_dims(pred_y, 0)

    n_traj_samples = pred_y.shape[0]
    pred_y = pred_y.reshape(n_traj_samples, -1)

    idx_not_nan = 1 - tf.math.is_nan(truth)
    if len(idx_not_nan) == 0:
        print("All are labels are NaNs!")
        ce_loss = tf.zeros_like([])

    pred_y = pred_y[:,idx_not_nan]
    truth = truth[idx_not_nan]

    if tf.reduce_sum(truth == 0.) == 0 or tf.reduce_sum(truth == 1.) == 0:
        print("Warning: all examples in a batch belong to the same class -- please
                increase the batch size.")

    assert(not tf.math.is_nan(pred_y).any())
    assert(not tf.math.is_nan(truth).any())

    # For each trajectory, we get n_traj_samples samples from z0 -- compute loss on
                all of them
    truth = tf.tile(truth.repeat, [n_traj_samples, 1])
    ce_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True)(pred_y, truth)

    # divide by number of patients in a batch
```

```
    ce_loss = ce_loss / n_traj_samples
    return ce_loss


def multiclass_ce(pred_y, true_label):

    if (len(pred_y.size()) == 3):
        pred_y = tf.expand_dims(pred_y,0)

    n_traj_samples, n_traj, n_tp, n_dims = pred_y.shape

    # For each trajectory, we get n_traj_samples samples from z0 -- compute loss on
            all of them
    true_label = true_label.repeat(n_traj_samples, 1, 1)

    pred_y = pred_y.reshape(n_traj_samples * n_traj * n_tp, n_dims)
    true_label = true_label.reshape(n_traj_samples * n_traj * n_tp, n_dims)

    if (pred_y.size(-1) > 1) and (true_label.size(-1) > 1):
        assert(pred_y.size(-1) == true_label.size(-1))
        # targets are in one-hot encoding -- convert to indices
        _, true_label = tf.reduce_max(true_label, -1)

    res = []
    for i in range(true_label.size(0)):
        pred = pred_y[i]
        labels = true_label[i]

        pred = tf.reshape(pred, (-1, n_dims))

        if (len(labels) == 0):
            continue

        ce_loss = tf.keras.losses.CategoricalCrossentropy()(pred, tf.cast(labels,
                tf.int64))
        res.append(ce_loss)

    ce_loss = tf.stack(res, 0)
    ce_loss = tf.mean(ce_loss)
    return ce_loss
```

misc.py

```
import six
import warnings
from typing import Iterable

from tensorflow.python import ops
import tensorflow as tf


def cast_double(x):
    if isinstance(x, Iterable):
        try:
            x = tf.cast(x, tf.float32)
        except Exception:
            xn = []
            for xi in x:
                xi = cast_double(xi)
                xn.append(xi)

            x = type(x)(xn)

    else:
        if hasattr(x, 'dtype') and x.dtype != tf.float32:
            x = tf.cast(x, tf.float32)

        elif type(x) != float:
            x = float(x)

    return x


def func_cast_double(func):
    """ Casts all Tensor arguments to float64 """

    @six.wraps(func)
    def wrapper(*args, **kwargs):
        cast_args = []
        for arg in args:
            if isinstance(arg, tf.Tensor) or isinstance(arg, tf.Variable):
                arg = cast_double(arg)
            elif type(arg) == tuple or type(tuple) == list:
```

```
                arg = cast_double(arg)

            cast_args.append(arg)

        result = func(*cast_args, **kwargs)
        return result

    return wrapper


def move_to_device(x, device):
    """ Utility function to move a tensor to a device """

    if device is None:
        return x

    # tf.Variables cannot be moved to a device
    if not isinstance(x, tf.Tensor):
        return x

    if isinstance(device, tf.Tensor):
        device = device.device

    # check if device is empty string
    if len(device) == 0:
        return x

    if '/' in device:
        device = device.replace('/', '')

    splits = device.split(':')[-2:]
    device, id = splits
    id = int(id)

    x_device = x.device.lower()

    if 'cpu' in device.lower() and 'cpu' not in x_device:
        x = x.cpu()

    elif 'gpu' in device.lower() and 'gpu' not in x_device:
        x = x.gpu(id)
```

```
    return x


def _check_len(x):
    """ Utility function to get the length of the tensor """
    if hasattr(x, 'shape'):
        return x.shape[0]
    else:
        return len(x)


def _numel(x, dtype=None):
    """ Compute number of elements in the input tensor """
    if dtype is None:
        dtype = x.dtype
    return tf.cast(tf.reduce_prod(x.shape), dtype)


def _is_floating_tensor(x):
    return x.dtype in [tf.float16, tf.float32, tf.float64]


def _flatten(sequence):
    flat = [tf.reshape(p, [-1]) for p in sequence]
    return tf.concat(flat, 0) if len(flat) > 0 else tf.convert_to_tensor([])


def _flatten_convert_none_to_zeros(sequence, like_sequence):
    flat = [
        tf.reshape(p, [-1]) if p is not None else tf.reshape(tf.zeros_like(q), [-1])
        for p, q in zip(sequence, like_sequence)
    ]
    return tf.concat(flat, 0) if len(flat) > 0 else tf.convert_to_tensor([])


def _flatten_recover(sequence):
    shapes = [p.shape for p in sequence]
    numels = [tf.cast(tf.reduce_prod(shape), tf.int32).numpy() for shape in shapes]

    flat = [tf.reshape(p, [-1]) for p in sequence]
```

```
    out = tf.concat(flat, 0) if len(flat) > 0 else tf.convert_to_tensor([])

    def _recover_shapes(flat):
        params_splits = tf.split(flat, numels)
        param_list = [tf.reshape(p, shape)
                      for p, shape in zip(params_splits, shapes)]

        return param_list

    return out, _recover_shapes


def _flatten_convert_none_to_zeros_recover(sequence, like_sequence):
    shapes = [p.shape if p is not None else q.shape for p, q in zip(sequence,
                like_sequence)]
    numels = [tf.cast(tf.reduce_prod(shape), tf.int32).numpy() for shape in shapes]

    flat = [
        tf.reshape(p, [-1]) if p is not None else tf.reshape(tf.zeros_like(q), [-1])
        for p, q in zip(sequence, like_sequence)
    ]
    out = tf.concat(flat, 0) if len(flat) > 0 else tf.convert_to_tensor([])

    def _recover_shapes(flat):
        params_splits = tf.split(flat, numels)
        param_list = [tf.reshape(p, shape)
                      for p, shape in zip(params_splits, shapes)]

        return param_list

    return out, _recover_shapes


def _possibly_nonzero(x):
    return isinstance(x, tf.Tensor) or x != 0


def _scaled_dot_product(scale, xs, ys):
    """Calculate a scaled, vector inner product between lists of Tensors."""
    # Using _possibly_nonzero lets us avoid wasted computation.
```

```
    return sum([(scale * x) * y for x, y in zip(xs, ys) if _possibly_nonzero(x) or
                _possibly_nonzero(y)])


def _dot_product(xs, ys):
    """Calculate the vector inner product between two lists of Tensors."""
    return sum([x * y for x, y in zip(xs, ys)])


def _has_converged(y0, y1, rtol, atol):
    """Checks that each element is within the error tolerance."""
    error_tol = tuple(atol + rtol * tf.maximum(tf.abs(y0_), tf.abs(y1_)) for y0_, y1_
                in zip(y0, y1))
    error = tuple(tf.abs(y0_ - y1_) for y0_, y1_ in zip(y0, y1))
    return all(tf.reduce_all(error_ < error_tol_) for error_, error_tol_ in zip(error,
                error_tol))


def _convert_to_tensor(a, dtype=None, device=None):
    if not isinstance(a, tf.Tensor):
        a = tf.convert_to_tensor(a)
    if dtype is not None:
        a = tf.cast(a, dtype)
    if device is not None:
        a = move_to_device(a, device)
    return a


def _is_finite(tensor):
    _check = tf.cast(tf.math.is_inf(tensor), tf.int64) + tf.cast(tf.math.is_nan(tensor),
                tf.int64)
    _check = tf.cast(_check, tf.bool)
    return not tf.reduce_any(_check)


def _decreasing(t):
    v = tf.reduce_all(t[1:] < t[:-1])
    return v


def _assert_increasing(t):
```

```
    assert tf.reduce_all(t[1:] > t[:-1]), 't must be strictly increasing or decreasing'


def _is_iterable(inputs):
    try:
        iter(inputs)
        return True
    except TypeError:
        return False


def _norm(x):
    """Compute RMS norm."""
    if isinstance(x, tf.Tensor):
        return tf.norm(x) / (_numel(x) ** 0.5)
    else:
        return tf.sqrt(sum(tf.norm(x_) ** 2 for x_ in x) / sum(_numel(x_) for x_ in x))


def _handle_unused_kwargs(solver, unused_kwargs):
    if len(unused_kwargs) > 0:
        warnings.warn('{}: Unexpected arguments
                {}'.format(solver.__class__.__name__, unused_kwargs))


def _select_initial_step(fun, t0, y0, order, rtol, atol, f0=None):
    """Empirically select a good initial step.
    The algorithm is described in [1]_.
    Parameters
    ----------
    fun : callable
        Right-hand side of the system.
    t0 : float
        Initial value of the independent variable.
    y0 : ndarray, shape (n,)
        Initial value of the dependent variable.
    direction : float
        Integration direction.
    order : float
        Method order.
    rtol : float
```

```
    Desired relative tolerance.
atol : float
    Desired absolute tolerance.
Returns
-------
h_abs : float
    Absolute value of the suggested initial step.
References
----------
.. [1] E. Hairer, S. P. Norsett G. Wanner, "Solving Ordinary Differential
       Equations I: Nonstiff Problems", Sec. II.4.
"""
t0 = move_to_device(t0, y0[0].device)
t0 = cast_double(t0)

y0 = cast_double(y0)

if f0 is None:
    f0 = fun(t0, y0)

f0 = cast_double(f0)

if hasattr(y0, 'shape'):
    count = y0.shape[0]
else:
    count = len(y0)

rtol = rtol if _is_iterable(rtol) else [rtol] * count
atol = atol if _is_iterable(atol) else [atol] * count

rtol = [cast_double(r) for r in rtol]
atol = [cast_double(a) for a in atol]

scale = tuple(atol_ + tf.abs(y0_) * rtol_ for y0_, atol_, rtol_ in zip(y0, atol, rtol))
scale = [cast_double(s) for s in scale]

d0 = tuple(_norm(y0_ / scale_) for y0_, scale_ in zip(y0, scale))
d1 = tuple(_norm(f0_ / scale_) for f0_, scale_ in zip(f0, scale))

if max(d0).numpy() < 1e-5 or max(d1).numpy() < 1e-5:
    h0 = move_to_device(tf.convert_to_tensor(1e-6), t0)
```

```
    else:
        h0 = 0.01 * max(d0_ / d1_ for d0_, d1_ in zip(d0, d1))

    h0 = cast_double(h0)

    y1 = tuple(y0_ + h0 * f0_ for y0_, f0_ in zip(y0, f0))
    f1 = fun(t0 + h0, y1)
    f1 = cast_double(f1)

    d2 = tuple(_norm((f1_ - f0_) / scale_) / h0 for f1_, f0_, scale_ in zip(f1, f0,
            scale))

    if max(d1).numpy() <= 1e-15 and max(d2).numpy() <= 1e-15:
        h1 = tf.reduce_max([move_to_device(tf.convert_to_tensor(1e-6,
            dtype=tf.float64), h0.device), h0 * 1e-3])
    else:
        h1 = (0.01 / max(d1 + d2)) ** (1. / float(order + 1))

    return tf.reduce_min([100 * h0, h1])


def _compute_error_ratio(error_estimate, error_tol=None, rtol=None, atol=None,
            y0=None, y1=None):
    if error_tol is None:
        assert rtol is not None and atol is not None and y0 is not None and y1 is not
            None
        rtol if _is_iterable(rtol) else [rtol] * len(y0)
        atol if _is_iterable(atol) else [atol] * len(y0)
        y0 = cast_double(y0)

        error_tol = tuple(
            atol_ + rtol_ * tf.reduce_max([tf.abs(y0_), tf.abs(y1_)])
            for atol_, rtol_, y0_, y1_ in zip(atol, rtol, y0, y1)
        )
    error_ratio = tuple(error_estimate_ / error_tol_ for error_estimate_, error_tol_ in
            zip(error_estimate, error_tol))
    mean_sq_error_ratio = tuple(tf.reduce_mean(error_ratio_ * error_ratio_) for
            error_ratio_ in error_ratio)
    return mean_sq_error_ratio
```

```
def _optimal_step_size(last_step, mean_error_ratio, safety=0.9, ifactor=10.0,
                       dfactor=0.2, order=5):
    """Calculate the optimal size for the next step."""
    mean_error_ratio = max(mean_error_ratio)  # Compute step size based on
                       highest ratio.

    if mean_error_ratio == 0:
        return last_step * ifactor

    if mean_error_ratio < 1:
        dfactor = _convert_to_tensor(1, dtype=tf.float32,
                       device=mean_error_ratio.device)

    error_ratio = tf.sqrt(mean_error_ratio)
    error_ratio = cast_double(error_ratio)
    error_ratio = move_to_device(error_ratio, last_step.device)

    exponent = tf.convert_to_tensor(1 / order)
    exponent = cast_double(exponent)
    exponent = move_to_device(exponent, last_step.device)

    factor = tf.reduce_max([1 / ifactor, tf.reduce_min([error_ratio ** exponent /
                       safety, 1 / dfactor])])
    return last_step / factor


def _check_inputs(func, y0, t):
    tensor_input = False
    if isinstance(y0, tf.Tensor):
        tensor_input = True
        if not isinstance(y0, ops.EagerTensor):
            warnings.warn('Input is *not* an EagerTensor ! '
                          'Dummy op with zeros will be performed instead.')

            y0 = tf.convert_to_tensor(tf.zeros(y0.shape))

        y0 = (y0,)
        _base_nontuple_func_ = func

        def base_nontuple_func(t, y):
            return (_base_nontuple_func_(t, y[0]),)
```

```
        func = base_nontuple_func

    if ((type(y0) == tuple) or (type(y0) == list)):
        if not tensor_input:
            y0_type = type(y0)
            y0 = list(y0)

            for i in range(len(y0)):
                assert isinstance(y0[i], tf.Tensor), 'each element must be a tf.Tensor ' \
                                                     'but received {}'.format(type(y0[i]))

                if not isinstance(y0[i], ops.EagerTensor):
                    warnings.warn('Input %d (zero-based) is *not* an EagerTensor ! '
                                  'Dummy op with zeros will be performed instead.' % (i))

                    y0[i] = tf.convert_to_tensor(tf.zeros(y0[i].shape))

            y0 = y0_type(y0)  # return to same type
    else:
        raise ValueError('y0 must be either a tf.Tensor or a tuple')

    if _decreasing(t):
        t = -t
        _base_reverse_func = func

        def base_reverese_func(t, y):
            return tuple(-f_ for f_ in _base_reverse_func(-t, y))
        func = base_reverese_func

    for y0_ in y0:
        if not tf.debugging.is_numeric_tensor(y0_):
            raise TypeError('`y0` must be a floating point Tensor but is a
                  {}'.format(y0_.dtype))
    if not tf.debugging.is_numeric_tensor(t):
        raise TypeError('`t` must be a floating point Tensor but is a
                  {}'.format(t.dtype))

    return tensor_input, func, y0, t
```

model.py

```
import tensorflow as tf
```

```
import tensorflow_probability as tfp
import numpy as np
from solver.losses import *
from solver.utils import split_last_dim

from solver.misc import (
    _handle_unused_kwargs, _select_initial_step, _convert_to_tensor,
                          _scaled_dot_product, _is_iterable,
    _optimal_step_size, _compute_error_ratio, move_to_device, cast_double
)

class RecognitionRNN(tf.keras.Model):
    def __init__(self, latent_dim=4, obs_dim=2, nhidden=25, nbatch=1):
        super().__init__()
        self.nhidden = nhidden
        self.nbatch = nbatch
        self.i2h = tf.keras.layers.Dense(nhidden, activation='tanh')
        self.h2o = tf.keras.layers.Dense(latent_dim * 2)

    #@tf.function
    def call(self, x, h):
        x = cast_double(x)
        h = cast_double(h)

        combined = tf.concat((x, h), axis=1)
        h = self.i2h(combined)
        out = self.h2o(h)
        return out, h

    def initHidden(self):
        return tf.zeros([self.nbatch, self.nhidden], dtype=tf.float32)

class Decoder(tf.keras.Model):
    def __init__(self, latent_dim=4, obs_dim=2, nhidden=20):
        super().__init__()
        self.fc1 = tf.keras.layers.Dense(obs_dim, activation=None)
        #self.fc2 = tf.keras.layers.Dense(obs_dim)

    def call(self, z):
        z = cast_double(z)
```

```
        out = self.fc1(z)
        #out = self.fc2(out)
        return out

class GRU_unit(tf.keras.models.Model):
    def __init__(self, latent_dim, input_dim,
        update_gate = None,
        reset_gate = None,
        new_state_net = None,
        n_units = 100):
        super(GRU_unit, self).__init__()

        if update_gate is None:
            #update gate input [latent_dim * 2 + input_dim]
            self.update_gate = tf.keras.models.Sequential([
                tf.keras.layers.Dense(units=n_units),
                tf.keras.layers.Activation('tanh'),
                tf.keras.layers.Dense(latent_dim),
                tf.keras.layers.Activation('sigmoid')])
        else:
            self.update_gate  = update_gate

        if reset_gate is None:
            self.reset_gate =tf.keras.models.Sequential([
                tf.keras.layers.Dense(units=n_units),
                tf.keras.layers.Activation('tanh'),
                tf.keras.layers.Dense(latent_dim),
                tf.keras.layers.Activation('sigmoid')])
        else:
            self.reset_gate  = reset_gate

        if new_state_net is None:
            self.new_state_net = tf.keras.models.Sequential([
                tf.keras.layers.Dense(units=n_units),
                tf.keras.layers.Activation('tanh'),
                tf.keras.layers.Dense(latent_dim*2)])
        else:
            self.new_state_net  = new_state_net

    @tf.function
    def call(self, y_mean, y_std, x):
```

```
        y_concat = tf.concat([y_mean, y_std, x],-1)

        update_gate = self.update_gate(y_concat)
        reset_gate = self.reset_gate(y_concat)
        concat = tf.concat([y_mean * reset_gate, y_std * reset_gate, x], -1)

        new_state, new_state_std = split_last_dim(self.new_state_net(concat))
        new_state_std = tf.abs(new_state_std)

        new_y = (1-update_gate) * new_state + update_gate * y_mean
        new_y_std = (1-update_gate) * new_state_std + update_gate * y_std
        new_y_std = tf.abs(new_y_std)
        return new_y, new_y_std

class RunningAverageMeter(object):
    """Computes and stores the average and current value"""

    def __init__(self, momentum=0.99):
        self.momentum = momentum
        self.reset()

    def reset(self):
        self.val = None
        self.avg = 0

    def update(self, val):
        if self.val is None:
            self.avg = val
        else:
            self.avg = self.avg * self.momentum + val * (1 - self.momentum)
        self.val = val

class VAEBaseline(tf.keras.models.Model):
    """
    Variational Autoencoder wrapper
    """
    def __init__(self, input_dim, latent_dim,
        z0_prior,
        obsrv_std = 0.01,
        use_binary_classif = False,
        classif_per_tp = False,
```

```
use_poisson_proc = False,
linear_classifier = False,
n_labels = 1,
train_classif_w_reconstr = False,
**kwargs):

dynamic = kwargs.pop('dynamic', True)

super().__init__(dynamic=dynamic)
print('dynamic - {}'.format(dynamic))
self.input_dim = input_dim
self.latent_dim = latent_dim
self.n_labels = n_labels

self.obsrv_std =  tf.convert_to_tensor([obsrv_std], dtype=tf.float32)

self.z0_prior = z0_prior
self.use_binary_classif = use_binary_classif
self.classif_per_tp = classif_per_tp
self.use_poisson_proc = use_poisson_proc
self.linear_classifier = linear_classifier
self.train_classif_w_reconstr = train_classif_w_reconstr

z0_dim = latent_dim
if use_poisson_proc:
   z0_dim += latent_dim

if use_binary_classif:
   if linear_classifier:
      self.classifier = tf.keras.models.Sequential[
         tf.keras.layers.Dense(n_labels,
                               kernel_initializer='random_normal',
                               bias_initializer='zeros')]
   else:
      self.classifier = tf.keras.models.Sequential[
         tf.keras.layers.Dense(n_labels,
                               kernel_initializer='random_normal',
                               bias_initializer='zeros'),
         tf.keras.layers.Activation('relu'),
         tf.keras.layers.Dense(300,
                           kernel_initializer='random_normal',
```

```
                        bias_initializer='zeros'),
            tf.keras.layers.Activation('relu'),
            tf.keras.layers.Dense(z0_dim,
                        kernel_initializer='random_normal',
                        bias_initializer='zeros')]

def get_gaussian_likelihood(self, truth, pred):
    # pred_y shape [n_traj_samples, n_traj, n_tp, n_dim]
    # truth shape  [n_traj, n_tp, n_dim]
    n_traj, n_tp, n_dim = truth.shape

    # Compute likelihood of the data under the predictions
    truth_repeated = tf.tile(tf.expand_dims(truth,0), [n_traj, 1, 1, 1])

    log_density_data = gaussian_log_density(pred, truth_repeated, self.obsrv_std)
    # shape: [n_traj_samples]
    return tf.reduce_sum(log_density_data, -1)

def get_mse(self, truth, pred):
    # pred_y shape [n_traj_samples, n_traj, n_tp, n_dim]
    # truth shape  [n_traj, n_tp, n_dim]
    n_traj, n_tp, n_dim = truth.shape

    # Compute likelihood of the data under the predictions
    truth_repeated = tf.tile(tf.expand_dims(truth,0), [n_traj, 1, 1, 1])
    log_density_data = mse(pred, truth_repeated)
    # shape: [n_traj_samples]
    return tf.reduce_mean(log_density_data)

def compute_all_losses(self, batch_dict, n_traj_samples = 1, kl_coef = 1.,
                       mode='extrap'):
    # Condition on subsampled points
    # Make predictions for all the points
    # Make sure that time is 1-D vector
    if (len(batch_dict["observed_tp"].shape)==2):
        batch_dict["observed_tp"] = tf.squeeze(batch_dict["observed_tp"])
    if (len(batch_dict["tp_to_predict"].shape)==2):
        batch_dict["tp_to_predict"] = tf.squeeze(batch_dict["tp_to_predict"])
    pred_y, info = self.get_reconstruction(batch_dict["tp_to_predict"],
                batch_dict["observed_data"],
                batch_dict["observed_tp"],
```

```
                n_traj_samples = n_traj_samples,
                mode = mode)

fp_mu, fp_std, fp_enc = info['first_point']
fp_distr = tfp.distributions.Normal(loc=fp_mu, scale=fp_std)

kl_div_z0 = tfp.distributions.kl_divergence(fp_distr, self.z0_prior)
# Mean over number of latent dimensions
# kldiv_z0 shape: [n_traj_samples, n_traj, n_latent_dims] if prior is a mixture
                    of gaussians (KL is estimated)
# kldiv_z0 shape: [1, n_traj, n_latent_dims] if prior is a standard gaussian (KL
                    is computed exactly)
# shape after: [n_traj_samples]
kl_div_z0 = tf.reduce_mean(kl_div_z0,(1,2))

# Compute likelihood of all the points
rec_likelihood = self.get_gaussian_likelihood(
                batch_dict["data_to_predict"],
                pred_y)

mse = self.get_mse(
      batch_dict["data_to_predict"],
      pred_y)

if self.use_poisson_proc:
   pois_log_likelihood = poisson_proc_likelihood(
      batch_dict["data_to_predict"], pred_y,
      info)
   # Take mean over n_traj
   pois_log_likelihood = tf.reduce_mean(pois_log_likelihood, 1)

ce_loss = tf.zeros_like([0.], tf.float32)
if ("labels" in batch_dict.keys()) and (batch_dict["labels"] is not None) and
                    self.use_binary_classif:

   if (batch_dict["labels"].shape[-1] == 1) or (len(batch_dict["labels"].shape)
                    == 1):
      ce_loss = binary_ce(
         info["label_predictions"],
         batch_dict["labels"])
   else:
```

```
            ce_loss = multiclass_ce(
                info["label_predictions"],
                batch_dict["labels"])


        # IWAE Loss
        loss = -tf.reduce_mean(rec_likelihood - kl_coef*kl_div_z0, 0)
        #loss = -tf.reduce_logsumexp(rec_likelihood - kl_coef*kl_div_z0, 0)
        if tf.math.is_nan(loss):
            loss = -tf.reduce_mean(rec_likelihood - kl_coef*kl_div_z0, 0)

        if self.use_poisson_proc:
            loss -= 0.1*pois_log_likelihood

        if self.use_binary_classif:
            if self.train_classif_w_reconstr:
                loss = loss +  ce_loss * 100
            else:
                loss =  ce_loss

        results = {}
        results["loss"] = tf.reduce_mean(loss)
        results["likelihood"] = tf.stop_gradient(tf.reduce_mean(rec_likelihood))
        results["mse"] = tf.stop_gradient(tf.reduce_mean(mse))
        results["cross_entropy"] = tf.stop_gradient(tf.reduce_mean(ce_loss))
        results["kl_first_p"] =  tf.stop_gradient(tf.reduce_mean(kl_div_z0))
        results["std_first_p"] = tf.stop_gradient(tf.reduce_mean(fp_std))

        if self.use_poisson_proc:
            results["pois_likelihood"] =
                        tf.stop_gradient(tf.reduce_mean(pois_log_likelihood))

        return results

class ODERNNEncoder(tf.keras.models.Model):
    def __init__(self, latent_dim, input_dim, z0_diffeq_solver = None,
                 z0_dim = None,
                 GRU_update=None,
                 n_gru_units = 100,
                 backwards_evaluation = True,
                 adjoint=True,
```

```
        save_info = False,
        **kwargs):

dynamic = kwargs.pop('dynamic', True)
super().__init__(**kwargs, dynamic=dynamic)
if z0_dim is None:
   self.z0_dim = latent_dim
else:
   self.z0_dim = z0_dim
self.z0_diffeq_solver = z0_diffeq_solver
self.latent_dim = latent_dim
self.input_dim = input_dim
self.backwards_evaluation = backwards_evaluation
self.save_info = save_info
self.adjoint = adjoint

if GRU_update is None:
   self.GRU_update = GRU_unit(self.latent_dim, self.input_dim,
                n_units=n_gru_units)
   #self.GRU_update = tf.keras.layers.GRU(units=self.latent_dim*2,
                stateful=True, return_sequences=True,
                bias_initializer='glorot_uniform')
else:
   self.GRU_update = GRU_update

#self.var_mean = tf.keras.Sequential([
#    #tf.keras.layers.InputLayer(input_shape=(n_inputs,)),
#    tf.keras.layers.Dense(self.latent_dim*2),
#    tfp.layers.DistributionLambda(
#        lambda t: tfp.distributions.Normal(loc=t[..., :self.latent_dim],
#                scale=1e-2 + tf.math.softplus(0.05 * t[..., self.latent_dim:]))),
#
#])

self.transform_z0 = tf.keras.models.Sequential([
   tf.keras.layers.Dense(100,kernel_initializer='random_normal',
                bias_initializer='zeros', activation=None),
   tf.keras.layers.Activation('tanh'),
   tf.keras.layers.Dense(self.z0_dim*2, kernel_initializer='random_normal',
                bias_initializer='zeros', activation=None)
])
```

```
#@tf.function
def call(self, data, time_steps):
    data, time_steps = tf.cast(data, tf.float32), tf.cast(time_steps, tf.float32)
    n_traj, n_tp, n_dims = data.shape
    backwards = self.backwards_evaluation

    save_info = self.save_info
    if len(time_steps) == 1:
        prev_y = tf.zeros((1, n_traj, self.latent_dim))
        prev_std = tf.zeros((1, n_traj, self.latent_dim))
        x_i = tf.expand_dims(data[:,0,:], 0)
        last_yi, last_std = self.GRU_update(prev_y, prev_std, x_i)
        #y_concat = tf.concat([prev_y, prev_std, x_i])
        #rec_out = self.GRU_update(y_concat)
        #probas = self.var_mean(rec_out)

        #last_yi, last_yi_std = probas.mean(), probas.stddev()
    else:
        last_yi, last_yi_std, _, extra_info = self.run_odernn(
            data, time_steps, run_backwards = backwards, save_info=save_info)
        means_z0 = tf.reshape(last_yi, [1, n_traj, self.latent_dim])
        std_z0 = tf.reshape(last_yi_std, [1, n_traj, self.latent_dim])

    mean_z0, std_z0 = split_last_dim(self.transform_z0(tf.concat([means_z0,
                        std_z0], -1)))
    std_z0 = tf.abs(std_z0)

    if save_info:
        self.extra_info = extra_info
    return mean_z0, std_z0

@tf.function
def run_odernn(self, data, time_steps,
    run_backwards = True, save_info=False):

    n_traj, n_tp, n_dims = data.shape

    t0 = time_steps[-1]
    if run_backwards:
        t0 = time_steps[0]
```

```
prev_y = tf.zeros((1, n_traj, self.latent_dim), tf.float32)
prev_std = tf.zeros((1, n_traj, self.latent_dim), tf.float32)
#print(time_steps)
prev_t, t_i = time_steps[-1] + 0.01,  time_steps[-1]

interval_length = time_steps[-1] - time_steps[0]
minimum_step = interval_length / 50

latent_ys = []
extra_info = []
# Run ODE backwards and combine the y(t) estimates using gating
time_points_iter = range(0, len(time_steps))
if run_backwards:
    time_points_iter = reversed(time_points_iter)

def max_step(thresh, prev_t, t_i, min_step):
    return max(2, np.ceil((prev_t - t_i) / minimum_step))

for t in time_points_iter:
    if (prev_t - t_i) < minimum_step:
        time_points = tf.stack((prev_t, t_i))
        #print('case 1 :{}'.format(prev_y))
        inc = self.z0_diffeq_solver.ode_func(prev_t, prev_y)*(t_i-prev_t)

        ode_sol = prev_y + inc
        ode_sol = tf.stack((prev_y, ode_sol), axis=2)
    else:
        #print(prev_t.numpy(), t_i.numpy(), minimum_step.numpy())
        n_intermediate_tp = tf.py_function(max_step, [2, prev_t, t_i,
                    minimum_step], Tout=tf.int32)
       # print('case 2 :'.format(prev_y))
        #n_intermediate_tp = max(2, tf.cast(((prev_t - t_i) /
                    minimum_step),tf.int32))

        time_points = tf.linspace(prev_t, t_i, n_intermediate_tp)
        ode_sol = self.z0_diffeq_solver(prev_y, time_points,
                    adjoint=self.adjoint)
    ode_sol = tf.cast(ode_sol, tf.float32)

    if tf.reduce_min(ode_sol[:,:,0,:]-prev_y)>=1e-3:
```

```
            print("Error: first point of the ODE is not equal to initial value")

        yi_ode = ode_sol[:, :, -1, :]
        x_i = tf.expand_dims(data[:,t,:], 0)
        #y_concat = tf.concat([yi_ode, prev_std, x_i], -1)
        last_yi, last_std = self.GRU_update(yi_ode, prev_std, x_i)
        #probas = self.var_mean(rec_out)
        #last_yi, last_std = probas.mean(), probas.stddev()
        prev_y, prev_std = last_yi, last_std
        prev_t, t_i = time_steps[t], time_steps[t-1]

        latent_ys.append(last_yi)
        if save_info:
            d = {"yi_ode": tf.stop_gradient(yi_ode),
                 "yi": tf.stop_gradient(last_yi), "yi_std": tf.stop_gradient(last_std),
                 "time_points": tf.stop_gradient(time_points), "ode_sol":
                        tf.stop_gradient(ode_sol)}
            extra_info.append(d)

    return last_yi, last_std, latent_ys, extra_info
```

odeint.py

```
from tensorflow.python.eager.context import eager_mode
import tensorflow as tf
#eager = tf.python.eager.context.eager_mode


from solver.adams import VariableCoefficientAdamsBashforth
from solver.dopri5 import Dopri5Solver
from solver.fixed_adams import AdamsBashforth, AdamsBashforthMoulton
from solver.fixed_grid import Euler, Midpoint, RK4, Huen
from solver.misc import _check_inputs
from solver.tsit5 import Tsit5Solver

SOLVERS = {
    'explicit_adams': AdamsBashforth,
    'fixed_adams': AdamsBashforthMoulton,
    'adams': VariableCoefficientAdamsBashforth,
    'tsit5': Tsit5Solver,
    'dopri5': Dopri5Solver,
    'euler': Euler,
```

```
    'midpoint': Midpoint,
    'rk4': RK4,
    'huen': Huen,
}


def odeint(func, y0, t, rtol=1e-7, atol=1e-9, method=None, options=None):
    """Integrate a system of ordinary differential equations.
    Solves the initial value problem for a non-stiff system of first order ODEs:
        ```
        dy/dt = func(t, y), y(t[0]) = y0
        ```
    where y is a Tensor of any shape.
    Output dtypes and numerical precision are based on the dtypes of the inputs
                            `y0`.
    Args:
        func: Function that maps a Tensor holding the state `y` and a scalar Tensor
            `t` into a Tensor of state derivatives with respect to time.
        y0: N-D Tensor giving starting value of `y` at time point `t[0]`. May
            have any floating point or complex dtype.
        t: 1-D Tensor holding a sequence of time points for which to solve for
            `y`. The initial time point should be the first element of this sequence,
            and each time must be larger than the previous time. May have any floating
            point dtype. Converted to a Tensor with float64 dtype.
        rtol: optional float64 Tensor specifying an upper bound on relative error,
            per element of `y`.
        atol: optional float64 Tensor specifying an upper bound on absolute error,
            per element of `y`.
        method: optional string indicating the integration method to use.
        options: optional dict of configuring options for the indicated integration
            method. Can only be provided if a `method` is explicitly set.
        name: Optional name for this operation.
    Returns:
        y: Tensor, where the first dimension corresponds to different
            time points. Contains the solved value of y for each desired time point in
            `t`, with the initial value `y0` being the first element along the first
            dimension.
    Raises:
        ValueError: if an invalid `method` is provided.
        TypeError: if `options` is supplied without `method`, or if `t` or `y0` has
            an invalid dtype.
```

```
    """
    with eager_mode():
    #with tf.python.eager.context.eager_mode():
        tensor_input, func, y0, t = _check_inputs(func, y0, t)

        if options is None:
            options = {}
        elif method is None:
            raise ValueError('cannot supply `options` without specifying `method`')

        if method is None:
            method = 'dopri5'

        solver = SOLVERS[method](func, y0, rtol=rtol, atol=atol, **options)
        solution = solver.integrate(t)

        if tensor_input:
            solution = solution[0]
        return solution
```

parse_dataset.py

```
import os

import numpy as np
import pandas as pd
import tensorflow as tf

from solver.utils import *

def csv_dataset(path=None, type='extrap', dataset=None, time_steps=None,
                        batch_size=64, tp_to_extrap=64, tp_to_interp=100):

    extrap_ = type=='extrap'

    def get_data_dict(data, timesteps, data_type='train'):
        # batch = tf.stack(batch)
        data_dict = {'data': data,
                'time_steps':timesteps}

        data_dict = split_and_subsample_batch(data_dict,data_type, extrap=extrap_)
        mode = data_dict['mode']
```

```
    data_dict = {k:v for k,v in data_dict.items() if k!='mode'}
    return data_dict, mode

batch_size = batch_size
time_points_to_extrap = tp_to_extrap

if path is not None:
    if os.path.exists(path):
        dataset = pd.read_csv(path)
        dataset = dataset.values
    else:
        raise FileNotFoundError()
elif dataset is not None:
    dataset = dataset
else:
    raise AttributeError('Either path or datset must be specified')

if (len(dataset.shape)==2):
    #we need to duplicate dataset in order to evaluate dynamics via test loop
    dataset = np.tile(np.expand_dims(dataset, axis=0),[2,1,1])
n_tp = dataset.shape[1]
n_traj = dataset.shape[0]

if time_steps is None:
    time_steps = np.linspace(0,1,n_tp).astype('float32')
    #time_steps = time_steps / len(time_steps)
else:
    time_steps = time_steps.astype('float32')


if type=='interp':
    # Creating dataset for interpolation
    n_reduced_tp = tp_to_interp
    # sample time points from different parts of the timeline,
    # so that the model learns from different parts of trajectory
    start_ind = np.random.randint(0, high=n_tp - n_reduced_tp +1, size=n_traj)
    end_ind = start_ind + n_reduced_tp
    sliced = []
    for i in range(n_traj):
        sliced.append(dataset[i, start_ind[i] : end_ind[i], :])
        dataset = sliced
```

```
        time_steps = time_steps[:n_reduced_tp]
    #dataset = tf.convert_to_tensor(dataset, tf.float32)
    #time_steps = tf.convert_to_tensor(time_steps)
    time_steps = np.expand_dims(time_steps, 0)
    train_y, test_y = split_train_test(dataset, train_fraq = 0.8)

    n_samples = len(dataset)
    input_dim = dataset.shape[-1]

    train_data, _ = get_data_dict(train_y, time_steps, data_type='train')
    test_data, mode = get_data_dict(test_y, time_steps, data_type='test')
    train_data = tf.data.Dataset.from_tensor_slices(train_data)
    test_data = tf.data.Dataset.from_tensor_slices(test_data)
    data_objects = {"train_dataloader":
                        inf_generator(train_data.cache().batch(batch_size).repeat()),
            "test_dataloader": inf_generator(test_data.batch(batch_size).repeat()),
            "input_dim": input_dim,
            'mode': mode,
            "n_train_batches": int(np.ceil(train_y.shape[1]/batch_size)),
            "n_test_batches": int(np.ceil(test_y.shape[1]/batch_size))}

    return data_objects
```

rk_common.py

```
# Based on
                    https://github.com/tensorflow/tensorflow/tree/master/tensorflow
                    /contrib/integrate
import collections
from solver.misc import _scaled_dot_product, _convert_to_tensor, cast_double,
                    func_cast_double

_ButcherTableau = collections.namedtuple('_ButcherTableau', 'alpha beta c_sol
                    c_error')

"""Saved state of the Runge Kutta solver.
    Attributes:
        y1: Tensor giving the function value at the end of the last time step.
        f1: Tensor giving derivative at the end of the last time step.
        t0: scalar float64 Tensor giving start of the last time step.
        t1: scalar float64 Tensor giving end of the last time step.
        dt: scalar float64 Tensor giving the size for the next time step.
        interp_coef: list of Tensors giving coefficients for polynomial
```

```
        interpolation between `t0` and `t1`.
"""
_RungeKuttaState = collections.namedtuple('_RungeKuttaState', 'y1, f1, t0, t1, dt,
                                          interp_coeff')


def _runge_kutta_step(func, y0, f0, t0, dt, tableau):
  """Take an arbitrary Runge-Kutta step and estimate error.
  Args:
    func: Function to evaluate like `func(t, y)` to compute the time derivative
      of `y`.
    y0: Tensor initial value for the state.
    f0: Tensor initial value for the derivative, computed from `func(t0, y0)`.
    t0: float64 scalar Tensor giving the initial time.
    dt: float64 scalar Tensor giving the size of the desired time step.
    tableau: optional _ButcherTableau describing how to take the Runge-Kutta
      step.
    name: optional name for the operation.
  Returns:
    Tuple `(y1, f1, y1_error, k)` giving the estimated function value after
    the Runge-Kutta step at `t1 = t0 + dt`, the derivative of the state at `t1`,
    estimated error at `t1`, and a list of Runge-Kutta coefficients `k` used for
    calculating these terms.
  """
  y0 = cast_double(y0)
  f0 = cast_double(f0)

  dtype = y0[0].dtype
  device = y0[0].device

  t0 = _convert_to_tensor(t0, dtype=dtype, device=device)
  dt = _convert_to_tensor(dt, dtype=dtype, device=device)

  k = tuple(map(lambda x: [x], f0))
  for alpha_i, beta_i in zip(tableau.alpha, tableau.beta):
    ti = t0 + alpha_i * dt
    yi = tuple(y0_ + _scaled_dot_product(dt, cast_double(beta_i), k_) for y0_, k_
                    in zip(y0, k))
    tuple(k_.append(cast_double(f_)) for k_, f_ in zip(k, func(ti, yi)))

  if not (tableau.c_sol[-1] == 0 and tableau.c_sol[:-1] == tableau.beta[-1]):
```

```
        # This property (true for Dormand-Prince) lets us save a few FLOPs.
        yi = tuple(y0_ + _scaled_dot_product(dt, tableau.c_sol, k_) for y0_, k_ in
                        zip(y0, k))

    y1 = yi
    f1 = tuple(k_[-1] for k_ in k)
    y1_error = tuple(_scaled_dot_product(dt, tableau.c_error, k_) for k_ in k)
    return (y1, f1, y1_error, k)


@func_cast_double
def rk4_step_func(func, t, dt, y, k1=None):
    if k1 is None:
        k1 = func(t, y)
    k1 = cast_double(k1)

    k2 = func(t + dt / 2, tuple(y_ + dt * k1_ / 2 for y_, k1_ in zip(y, k1)))
    k3 = func(t + dt / 2, tuple(y_ + dt * k2_ / 2 for y_, k2_ in zip(y, k2)))
    k4 = func(t + dt, tuple(y_ + dt * k3_ for y_, k3_ in zip(y, k3)))
    return tuple((k1_ + 2 * k2_ + 2 * k3_ + k4_) * (dt / 6) for k1_, k2_, k3_, k4_ in
                        zip(k1, k2, k3, k4))


@func_cast_double
def rk4_alt_step_func(func, t, dt, y, k1=None):
    """Smaller error with slightly more compute."""
    if k1 is None:
        k1 = func(t, y)
    k1 = cast_double(k1)

    k2 = func(t + dt / 3, tuple(y_ + dt * k1_ / 3 for y_, k1_ in zip(y, k1)))
    k2 = cast_double(k2)

    k3 = func(t + dt * 2 / 3, tuple(y_ + dt * (k1_ / -3 + k2_) for y_, k1_, k2_ in zip(y,
                        k1, k2)))
    k3 = cast_double(k3)

    k4 = func(t + dt, tuple(y_ + dt * (k1_ - k2_ + k3_) for y_, k1_, k2_, k3_ in zip(y,
                        k1, k2, k3)))
    k4 = cast_double(k4)
```

```
    return tuple((k1_ + 3 * k2_ + 3 * k3_ + k4_) * (dt / 8) for k1_, k2_, k3_, k4_ in
                 zip(k1, k2, k3, k4))
```

solvers.py

```
import abc

import tensorflow as tf

from solver.misc import (_assert_increasing, _handle_unused_kwargs,
                         move_to_device, cast_double, func_cast_double)


class AdaptiveStepsizeODESolver(object):
    __metaclass__ = abc.ABCMeta

    def __init__(self, func, y0, atol, rtol, **unused_kwargs):
        _handle_unused_kwargs(self, unused_kwargs)
        del unused_kwargs

        self.func = func
        self.y0 = y0
        self.atol = atol
        self.rtol = rtol

    def before_integrate(self, t):
        pass

    @abc.abstractmethod
    def advance(self, next_t):
        raise NotImplementedError

    def integrate(self, t):
        #print('in integrate')
        #print('t = {}'.format(t))
        _assert_increasing(t)
        solution = [cast_double(self.y0)]
        t = move_to_device(tf.cast(t, tf.float32), self.y0[0].device)
        self.before_integrate(t)
        for i in range(1, t.shape[0]):
            y = self.advance(t[i])
            y = cast_double(y)
```

```
            solution.append(y)
        return tuple(map(tf.stack, tuple(zip(*solution))))


class FixedGridODESolver(object):
    __metaclass__ = abc.ABCMeta

    def __init__(self, func, y0, step_size=None, grid_constructor=None,
                         **unused_kwargs):
        unused_kwargs.pop('rtol', None)
        unused_kwargs.pop('atol', None)
        _handle_unused_kwargs(self, unused_kwargs)
        del unused_kwargs

        self.func = func
        self.y0 = y0

        if step_size is not None and grid_constructor is None:
            self.grid_constructor = self._grid_constructor_from_step_size(step_size)
        elif grid_constructor is None:
            self.grid_constructor = lambda f, y0, t: t
        else:
            raise ValueError("step_size and grid_constructor are exclusive
                        arguments.")

    def _grid_constructor_from_step_size(self, step_size):

        def _grid_constructor(func, y0, t):
            start_time = t[0]
            end_time = t[-1]

            niters = tf.ceil((end_time - start_time) / step_size + 1).item()
            t_infer = move_to_device(tf.range(0, niters), t) * step_size + start_time
            if t_infer[-1] > t[-1]:
                t_infer[-1] = t[-1]

            return t_infer

        return _grid_constructor

    @property
```

```
@abc.abstractmethod
def order(self):
   pass

@abc.abstractmethod
def step_func(self, func, t, dt, y):
   pass

def integrate(self, t):
   _assert_increasing(t)
   t = tf.cast(t, self.y0[0].dtype)
   time_grid = self.grid_constructor(self.func, self.y0, t)
   assert tf.equal(time_grid[0], t[0]) and tf.equal(time_grid[-1], t[-1])
   time_grid = move_to_device(time_grid, self.y0[0].device)

   solution = [cast_double(self.y0)]

   j = 1
   y0 = cast_double(self.y0)
   for t0, t1 in zip(time_grid[:-1], time_grid[1:]):
      dy = self.step_func(self.func, t0, t1 - t0, y0)
      y1 = tuple(y0_ + dy_ for y0_, dy_ in zip(y0, dy))

      while j < t.shape[0] and t1 >= t[j]:
         y = self._linear_interp(t0, t1, y0, y1, t[j])
         solution.append(y)
         j += 1
      y0 = y1

   return tuple(map(tf.stack, tuple(zip(*solution))))

@func_cast_double
def _linear_interp(self, t0, t1, y0, y1, t):
   if t == t0:
      return y0
   if t == t1:
      return y1
   t0 = move_to_device(t0, y0[0].device)
   t1 = move_to_device(t1, y0[0].device)
   t = move_to_device(t, y0[0].device)
   slope = tuple((y1_ - y0_) / (t1 - t0) for y0_, y1_, in zip(y0, y1))
```

```
        return tuple(y0_ + slope_ * (t - t0) for y0_, slope_ in zip(y0, slope))
```

tsit5.py

```
import tensorflow as tf

from solver.misc import (
    _scaled_dot_product, _convert_to_tensor, _is_finite, _select_initial_step,
                        _handle_unused_kwargs,
    _numel, cast_double)
from solver.rk_common import _RungeKuttaState, _ButcherTableau,
                        _runge_kutta_step
from solver.solvers import AdaptiveStepsizeODESolver

# Parameters from [Tsitouras
                        (2011)](http://users.ntua.gr/tsitoura/RK54_new_v2.pdf)
_TSITOURAS_TABLEAU = _ButcherTableau(
    alpha=[0.161, 0.327, 0.9, 0.9800255409045097, 1., 1.],
    beta=[
        [0.161],
        [-0.008480655492357, 0.3354806554923570],
        [2.897153057105494, -6.359448489975075, 4.362295432869581],
        [5.32586482843925895, -11.74888356406283, 7.495539342889836, -
                        0.09249506636175525],
        [5.86145544294642038, -12.92096931784711, 8.159367898576159, -
                        0.071584973281401006, -0.02826905039406838],
        [0.09646076681806523, 0.01, 0.4798896504144996, 1.379008574103742, -
                        3.290069515436081, 2.324710524099774],
    ],
    c_sol=[0.09646076681806523, 0.01, 0.4798896504144996,
                        1.379008574103742, -3.290069515436081,
                        2.324710524099774, 0],
    c_error=[
        0.09646076681806523 - 0.001780011052226,
        0.01 - 0.000816434459657,
        0.4798896504144996 - -0.007880878010262,
        1.379008574103742 - 0.144711007173263,
        -3.290069515436081 - -0.582357165452555,
        2.324710524099774 - 0.458082105929187,
        -1 / 66,
    ],
)
```

```
def _interp_coeff_tsit5(t0, dt, eval_t):
    t = cast_double((eval_t - t0) / dt)
    b1 = -1.0530884977290216 * t * (t - 1.3299890189751412) * (t ** 2 -
                        1.4364028541716351 * t + 0.7139816917074209)
    b2 = 0.1017 * t ** 2 * (t ** 2 - 2.1966568338249754 * t +
                        1.2949852507374631)
    b3 = 2.490627285651252793 * t ** 2 * (t ** 2 - 2.38535645472061657 * t +
                        1.57803468208092486)
    b4 = -16.54810288924490272 * (t - 1.21712927295533244) * (t -
                        0.61620406037800089) * t ** 2
    b5 = 47.37952196281928122 * (t - 1.203071208372362603) * (t -
                        0.658047292653547382) * t ** 2
    b6 = -34.87065786149660974 * (t - 1.2) * (t - 0.666666666666666667) * t ** 2
    b7 = 2.5 * (t - 1) * (t - 0.6) * t ** 2
    return [b1, b2, b3, b4, b5, b6, b7]


def _interp_eval_tsit5(t0, t1, k, eval_t):
    dt = cast_double(t1 - t0)
    y0 = tuple(k_[0] for k_ in k)
    interp_coeff = _interp_coeff_tsit5(t0, dt, eval_t)
    y_t = tuple(y0_ + _scaled_dot_product(dt, interp_coeff, k_) for y0_, k_ in
                        zip(y0, k))
    return y_t


def _optimal_step_size(last_step, mean_error_ratio, safety=0.9, ifactor=10.0,
                        dfactor=0.2, order=5):
    """Calculate the optimal size for the next Runge-Kutta step."""
    if mean_error_ratio == 0:
        return last_step * ifactor
    if mean_error_ratio < 1:
        dfactor = _convert_to_tensor(1, dtype=tf.float32,
                        device=mean_error_ratio.device)
    error_ratio = tf.cast(mean_error_ratio, last_step.dtype)
    exponent = tf.convert_to_tensor(1 / order, dtype=last_step.dtype)
    factor = tf.maximum(1. / ifactor, tf.minimum((error_ratio ** exponent) / safety,
                        1 / dfactor))
    return last_step / factor
```

```
def _abs_square(x):
    return x * x


class Tsit5Solver(AdaptiveStepsizeODESolver):

    def __init__(
            self, func, y0, rtol, atol, first_step=None, safety=0.9, ifactor=10.0,
                        dfactor=0.2,
            max_num_steps=2 ** 31 - 1,
            **unused_kwargs
    ):
        _handle_unused_kwargs(self, unused_kwargs)
        del unused_kwargs

        self.func = func
        self.y0 = y0
        self.rtol = rtol
        self.atol = atol
        self.first_step = first_step
        self.safety = _convert_to_tensor(safety, dtype=tf.float32,
                        device=y0[0].device)
        self.ifactor = _convert_to_tensor(ifactor, dtype=tf.float32,
                        device=y0[0].device)
        self.dfactor = _convert_to_tensor(dfactor, dtype=tf.float32,
                        device=y0[0].device)
        self.max_num_steps = _convert_to_tensor(max_num_steps, dtype=tf.int32,
                        device=y0[0].device)
        self.order = 5

    def before_integrate(self, t):
        if self.first_step is None:
            first_step = _convert_to_tensor(_select_initial_step(self.func, t[0], self.y0,
                        4, self.rtol, self.atol),
                                device=t.device)
        else:
            first_step = _convert_to_tensor(0.01, dtype=t.dtype, device=t.device)
        self.rk_state = _RungeKuttaState(
            self.y0,
```

```
        cast_double(self.func(t[0], self.y0)), t[0], t[0], first_step,
        tuple(map(lambda x: [x] * 7, self.y0))
    )

def advance(self, next_t):
    """Interpolate through the next time point, integrating as necessary."""
    n_steps = 0
    while next_t > self.rk_state.t1:
        assert n_steps < self.max_num_steps, 'max_num_steps exceeded
                    ({}>={})'.format(n_steps, self.max_num_steps)
        self.rk_state = self._adaptive_tsit5_step(self.rk_state)
        n_steps += 1
    return _interp_eval_tsit5(self.rk_state.t0, self.rk_state.t1,
                    self.rk_state.interp_coeff, next_t)

def _adaptive_tsit5_step(self, rk_state):
    """Take an adaptive Runge-Kutta step to integrate the ODE."""
    y0, f0, _, t0, dt, _ = rk_state
    ####################################################
    #                   Assertions                     #
    ####################################################
    assert t0 + dt > t0, 'underflow in dt {}'.format(dt.numpy())
    for y0_ in y0:
        assert _is_finite(tf.abs(y0_)), 'non-finite values in state `y`: {}'.format(y0_)
    y1, f1, y1_error, k = _runge_kutta_step(self.func, y0, f0, t0, dt,
                    tableau=_TSITOURAS_TABLEAU)

    ####################################################
    #                   Error Ratio                    #
    ####################################################
    error_tol = tuple(self.atol + self.rtol * tf.reduce_max([tf.abs(y0_),
                    tf.abs(y1_)])
                for y0_, y1_ in zip(y0, y1))
    tensor_error_ratio = tuple(y1_error_ / error_tol_
                        for y1_error_, error_tol_ in zip(y1_error, error_tol))
    sq_error_ratio = tuple(
        tf.multiply(tensor_error_ratio_, tensor_error_ratio_)
        for tensor_error_ratio_ in tensor_error_ratio
    )
    mean_error_ratio = (
        sum(tf.reduce_sum(sq_error_ratio_)
```

```
            for sq_error_ratio_ in sq_error_ratio) / sum(_numel(sq_error_ratio_)
                                                        for sq_error_ratio_ in sq_error_ratio)
        )
        accept_step = mean_error_ratio <= 1

        ##########################################################
        #                Update RK State                 #
        ##########################################################
        y_next = y1 if accept_step else y0
        f_next = f1 if accept_step else f0
        t_next = t0 + dt if accept_step else t0
        dt_next = _optimal_step_size(dt, mean_error_ratio, self.safety, self.ifactor,
                          self.dfactor, order=self.order)
        k_next = k if accept_step else self.rk_state.interp_coeff
        rk_state = _RungeKuttaState(y_next, f_next, t0, t_next, dt_next, k_next)

        return rk_state
```

utils.py

```
import os
import logging
import pickle

import tensorflow as tf
import tensorflow_probability as tfp
import numpy as np
import pandas as pd
import math
import glob
import re
from shutil import copyfile
import sklearn as sk
import sklearn.metrics as skm
import subprocess
import datetime

def makedirs(dirname):
    if not os.path.exists(dirname):
```

```
        os.makedirs(dirname)

def save_checkpoint(state, save, epoch):
    if not os.path.exists(save):
        os.makedirs(save)
    filename = os.path.join(save, 'checkpt-%04d.h5' % epoch)
    tf.keras.models.save_model(model=state, filepath=filename)

def get_logger(logpath, filepath, package_files=[],
               displaying=True, saving=True, debug=False):
    logger = logging.getLogger()
    if debug:
        level = logging.DEBUG
    else:
        level = logging.INFO
    logger.setLevel(level)
    if saving:
        info_file_handler = logging.FileHandler(logpath, mode='w')
        info_file_handler.setLevel(level)
        logger.addHandler(info_file_handler)
    if displaying:
        console_handler = logging.StreamHandler()
        console_handler.setLevel(level)
        logger.addHandler(console_handler)
    logger.info(filepath)

    for f in package_files:
        logger.info(f)
        with open(f, 'r') as package_f:
            logger.info(package_f.read())

    return logger

def inf_generator(iterable):
    """Allows training with tf RepeatDataset in a single infinite loop:
        for i, (x, y) in enumerate(inf_generator(train_loader)):
    """
    iterator = iterable.__iter__()
    while True:
        try:
            yield iterator.__next__()
```

```
        except StopIteration:
            iterator = iterable.__iter__()

def dump_pickle(data, filename):
    with open(filename, 'wb') as pkl_file:
        pickle.dump(data, pkl_file)

def load_pickle(filename):
    with open(filename, 'rb') as pkl_file:
        filecontent = pickle.load(pkl_file)
    return filecontent

def split_last_dim(data):
    last_dim = data.shape[-1]
    last_dim = last_dim//2
    if len(data.shape) == 3:
        res = data[:,:,:last_dim], data[:,:,last_dim:]

    if len(data.shape) == 2:
        res = data[:,:last_dim], data[:,last_dim:]
    return res

def flatten(x, dim):
    return tf.reshape(x, (x.shape[:dim] + (-1, )))

def subsample_timepoints(data, time_steps, n_tp_to_sample = None):
    # n_tp_to_sample: number of time points to subsample. If not None, sample
exactly n_tp_to_sample points
    if n_tp_to_sample is None:
        return data, time_steps
    n_tp_in_batch = len(time_steps)

    if n_tp_to_sample > 1:
        # Subsample exact number of points
        assert(n_tp_to_sample <= n_tp_in_batch)
        n_tp_to_sample = int(n_tp_to_sample)

        for i in range(data.shape[0]):
            missing_idx = sorted(np.random.choice(np.arange(n_tp_in_batch),
n_tp_in_batch - n_tp_to_sample, replace = False))
            data[i, missing_idx] = 0.
```

```
    elif (n_tp_to_sample <= 1) and (n_tp_to_sample > 0):
        # Subsample percentage of points from each time series
        percentage_tp_to_sample = n_tp_to_sample
        for i in range(data.shape[0]):
            n_to_sample = int(n_tp_in_batch * percentage_tp_to_sample)
            subsampled_idx = sorted(np.random.choice(time_steps, n_to_sample,
replace = False))
            tp_to_set_to_zero = np.setdiff1d(time_steps, subsampled_idx)

            data[i, tp_to_set_to_zero] = 0.

    return data, time_steps

def cut_out_timepoints(data, time_steps, n_points_to_cut = None):
    # n_points_to_cut: number of consecutive time points to cut out
    if n_points_to_cut is None:
        return data, time_steps
    n_tp_in_batch = len(time_steps)
    if n_points_to_cut < 1:
        raise Exception("Number of time points to cut out must be > 1")

    assert(n_points_to_cut <= n_tp_in_batch)
    n_points_to_cut = int(n_points_to_cut)
    for i in range(data.shape[0]):
        start = np.random.choice(np.arange(5, n_tp_in_batch - n_points_to_cut-5),
replace = False)
        data[i, start : (start + n_points_to_cut)] = 0.
    return data, time_steps

def split_train_test(data, train_fraq = 0.8):
    n_samples = data.shape[0]
    data_train = data[:int(n_samples * train_fraq)]
    data_test = data[int(n_samples * train_fraq):]
    return data_train, data_test

def split_train_test_data_and_time(data, time_steps, train_fraq = 0.8):
    n_samples = data.shape[0]
    data_train = data[:int(n_samples * train_fraq)]
    data_test = data[int(n_samples * train_fraq):]

    assert(len(time_steps.shape) == 2)
```

```
    train_time_steps = time_steps[:, :int(n_samples * train_fraq)]
    test_time_steps = time_steps[:, int(n_samples * train_fraq):]

    return data_train, data_test, train_time_steps, test_time_steps

def get_next_batch(data):
    # Make the union of all time points and perform normalization across the whole
dataset
    data_dict = data.__next__()
    batch_dict = get_dict_template()
    batch_dict = {k:v for k,v in batch_dict.items() if k in data_dict.keys()}
    # remove the time points where there are no observations in this batch
    non_missing_tp = tf.reduce_sum(data_dict["observed_data"],(0,2)) != 0.
    batch_dict["observed_data"] = tf.boolean_mask(data_dict["observed_data"],
non_missing_tp, axis=1)
    batch_dict["observed_tp"] =
tf.boolean_mask(tf.squeeze(data_dict["observed_tp"]),non_missing_tp)

    batch_dict[ "data_to_predict"] = data_dict["data_to_predict"]
    batch_dict["tp_to_predict"] =  tf.squeeze(data_dict["tp_to_predict"])

    non_missing_tp = tf.reduce_sum(data_dict["data_to_predict"],(0,2)) != 0.
    batch_dict["data_to_predict"] = tf.boolean_mask(data_dict["data_to_predict"],
non_missing_tp, axis=1)
    batch_dict["tp_to_predict"] =
tf.boolean_mask(tf.squeeze(data_dict["tp_to_predict"]),non_missing_tp)

    if ("labels" in data_dict) and (data_dict["labels"] is not None):
        batch_dict["labels"] = data_dict["labels"]

    #batch_dict["mode"] = data_dict["mode"]
    return batch_dict

def get_h5_model(h5_path):
    if not os.path.exists(h5_path):
        raise Exception("Checkpoint " + h5_path + " does not exist.")
    # Load checkpoint.
    checkpt = tf.keras.models.load_model(h5_path)
    return checkpt
```

```
def update_learning_rate_optimizer(optimizer, initial_lr=0.1, decay_steps=10,
lowest = 1e-3, **kwargs):
   # initial_lr: initial learning rate
   # decay_steps: number of steps to perform lr decay until lowest reached
   schedule = tf.keras.optimizers.schedules.InverseTimeDecay(initial_lr,
                                          decay_steps,
                                          lowest)
   params_dict = {'learning_rate': schedule}
   params_dict.update(kwargs)
   return optimizer(**kwargs)

def reverse(tensor):
   idx = [i for i in range(tensor.shape[0]-1, -1, -1)]
   return tensor[idx]

def create_net(n_inputs, n_outputs, n_layers = 1,
   n_units = 100, nonlinear = tf.keras.activations.relu):
   model = [#tf.keras.layers.InputLayer(input_shape=n_inputs, ragged=True),
         tf.keras.layers.Dense(units=n_units)]
   for i in range(n_layers):
      #model.add(tf.keras.layers.Activation(nonlinear))
      model.append(tf.keras.layers.Dense(n_units, activation=nonlinear))
   #model.add(tf.keras.layers.Activation(nonlinear))
   model.append(tf.keras.layers.Dense(n_outputs, activation=nonlinear))
   return tf.keras.models.Sequential(model)

def get_item_from_pickle(pickle_file, item_name):
   from_pickle = load_pickle(pickle_file)
   if item_name in from_pickle:
      return from_pickle[item_name]
   return None

def get_dict_template():
   return {"observed_data": None,
         "observed_tp": None,
         "data_to_predict": None,
         "tp_to_predict": None,
         "labels": None
         }

def normalize_data(data):
```

```
    reshaped = data.reshape(-1, data.shape[-1])

    att_min = tf.reduce_min(reshaped, 0)[0]
    att_max = tf.reduce_max(reshaped, 0)[0]

    # we don't want to divide by zero
    att_max[ att_max == 0.] = 1.

    if (att_max != 0.).all():
        data_norm = (data - att_min) / att_max
    else:
        raise Exception("division by zero")

    if tf.math.is_nan(data_norm).numpy().any():
        raise Exception("data contains nans")

    return data_norm, att_min, att_max

def split_data_extrap(data_dict):
    n_observed_tp = data_dict["data"].shape[1] // 2

    split_dict = {"observed_data": tf.identity(data_dict["data"][:,:n_observed_tp,:]),
                  "observed_tp": tf.identity(data_dict["time_steps"][:,:n_observed_tp]),
                  "data_to_predict": tf.identity(data_dict["data"][:,n_observed_tp:,:]),
                  "tp_to_predict": tf.identity(data_dict["time_steps"][:,n_observed_tp:])}

    if ("labels" in data_dict):
        split_dict["labels"] = tf.identity(data_dict["labels"])

    split_dict["mode"] = "extrap"
    return split_dict

def split_data_interp(data_dict):
    split_dict = {"observed_data": tf.identity(data_dict["data"]),
                  "observed_tp": tf.identity(data_dict["time_steps"]),
                  "data_to_predict": tf.identity(data_dict["data"]),
                  "tp_to_predict": tf.identity(data_dict["time_steps"])}

    if ("labels" in data_dict):
        split_dict["labels"] = tf.identity(data_dict["labels"])
```

```
    split_dict["mode"] = "interp"
    return split_dict

def subsample_observed_data(data_dict, n_tp_to_sample = None, n_points_to_cut
= None):
    # n_tp_to_sample -- if not None, randomly subsample the time points. The
resulting timeline has n_tp_to_sample points
    # n_points_to_cut -- if not None, cut out consecutive points on the timeline.  The
resulting timeline has (N - n_points_to_cut) points

    if n_tp_to_sample is not None:
        # Randomly subsample time points
        data, time_steps = subsample_timepoints(
            tf.identity(data_dict["observed_data"]),
            time_steps = tf.identity(data_dict["observed_tp"]),
            n_tp_to_sample = n_tp_to_sample)

    if n_points_to_cut is not None:
        # Remove consecutive time points
        data, time_steps= cut_out_timepoints(
            tf.identity(data_dict["observed_data"]),
            time_steps = tf.identity(data_dict["observed_tp"]),
            n_points_to_cut = n_points_to_cut)

    new_data_dict = {}
    for key in data_dict.keys():
        new_data_dict[key] = data_dict[key]

    new_data_dict["observed_data"] = tf.identity(data)
    new_data_dict["observed_tp"] = tf.identity(time_steps)

    if n_points_to_cut is not None:
        # Cut the section in the data to predict as well
        # Used only for the demo on the periodic function
        new_data_dict["data_to_predict"] = tf.identity(data)
        new_data_dict["tp_to_predict"] = tf.identity(time_steps)

    return new_data_dict


def split_and_subsample_batch(data_dict, data_type = "train",
```

```
                        extrap=False, sample_tp=None, cut_tp=None):
    if data_type == "train":
        # Training set
        if extrap:
            processed_dict = split_data_extrap(data_dict)
        else:
            processed_dict = split_data_interp(data_dict)
    else:
        # Test set
        if extrap:
            processed_dict = split_data_extrap(data_dict)
        else:
            processed_dict = split_data_interp(data_dict)

    # Subsample points or cut out the whole section of the timeline
    if (sample_tp is not None) or (cut_tp is not None):
        processed_dict = subsample_observed_data(processed_dict,
            n_tp_to_sample = sample_tp,
            n_points_to_cut = cut_tp)
    # if (sample_tp is not None):
    #processed_dict = subsample_observed_data(processed_dict,
    #n_tp_to_sample = sample_tp)
    return processed_dict

def compute_loss_all_batches(model,
    test_dataloader, mode, dataset,
    n_batches, experimentID, device,
    n_traj_samples = 1, kl_coef = 1.,
    max_samples_for_eval = None):

    total = {}
    total["loss"] = 0
    total["likelihood"] = 0
    total["mse"] = 0
    total["kl_first_p"] = 0
    total["std_first_p"] = 0
    total["pois_likelihood"] = 0
    total["ce_loss"] = 0

    n_test_batches = 0
```

```
    classif_predictions = tf.zeros_like([])
    all_test_labels = tf.zeros_like([])

    for i in range(n_batches):
        print("Computing loss for batch {}".format(i))

        batch_dict = get_next_batch(test_dataloader)

        results  = model.compute_all_losses(batch_dict,
            n_traj_samples = n_traj_samples, kl_coef = kl_coef)

        if mode=='classification':
            n_labels = model.n_labels #batch_dict["labels"].size(-1)
            n_traj_samples = results["label_predictions"].shape[0]

            classif_predictions = tf.concat([classif_predictions,
                tf.reshape(results["label_predictions"],[n_traj_samples, -1, n_labels])],1)
            all_test_labels = tf.concat([all_test_labels,
                tf.reshape(batch_dict["labels"], [-1, n_labels])],0)

            for key in total.keys():
                if key in results:
                    var = results[key]
                if isinstance(var, tf.Tensor):
                    var = tf.stop_gradient(var)
                total[key] += var

        n_test_batches += 1

    if n_test_batches > 0:
        for key, value in total.items():
            total[key] = total[key] / n_test_batches

    if mode=='classification':
        #all_test_labels = all_test_labels.reshape(-1)
        # For each trajectory, we get n_traj_samples samples from z0 -- compute loss
on all of them
        all_test_labels = tf.tile(all_test_labels, [n_traj_samples,1,1])



        idx_not_nan = 1 - tf.math.is_nan(all_test_labels)
```

```
        classif_predictions = classif_predictions[idx_not_nan]
        all_test_labels = all_test_labels[idx_not_nan]

        dirname = "plots/" + str(experimentID) + "/"
        os.makedirs(dirname, exist_ok=True)

        total["auc"] = 0.
        if tf.reduce_sum(all_test_labels) != 0.:
            print("Number of labeled examples:
{}".format(len(all_test_labels.reshape(-1))))
            print("Number of examples with mortality 1:
{}".format(tf.reduce_sum(all_test_labels == 1.)))

            # Cannot compute AUC with only 1 class
            total["auc"] = tf.py_function(skm.roc_auc_score,
[all_test_labels.numpy().reshape(-1),
                classif_predictions.numpy().reshape(-1)], Tout=tf.float32)
        else:
            print("Warning: Couldn't compute AUC -- all examples are from the same
class")
    return total
```

ode_block.py

```
import tensorflow as tf
import tensorflow_probability as tfp

from solver.model import Decoder, ODERNNEncoder, VAEBaseline
from solver.odeint import odeint
from solver.adjoint import odeint_adjoint

MAX_NUM_STEPS = 1000


class ODEFunc(tf.keras.models.Model):

    def __init__(self, hidden_dim, augment_dim=0,
                 time_dependent=False, non_linearity='relu',
                 **kwargs):
        """
        MLP modeling the derivative of ODE system.
        # Arguments:
```

```
        hidden_dim : int
            Dimension of hidden layers.
        augment_dim: int
            Dimension of augmentation. If 0 does not augment ODE,
            otherwise augments
            it with augment_dim dimensions.
        time_dependent : bool
            If True adds time as input, making ODE time dependent.
        non_linearity : string
            One of 'relu' and 'softplus'
    """

    dynamic = kwargs.pop('dynamic', True)
    super().__init__(**kwargs, dynamic=dynamic)
    self.augment_dim = augment_dim
    # self.data_dim = input_dim
    # self.input_dim = input_dim + augment_dim
    self.hidden_dim = hidden_dim
    self.nfe = 0  # Number of function evaluations
    self.time_dependent = time_dependent

    self.fc1 = tf.keras.layers.Dense(hidden_dim)
    self.fc2 = tf.keras.layers.Dense(hidden_dim)

    self.fc3 = None

    if non_linearity == 'relu':
        self.non_linearity = tf.keras.layers.ReLU()
    elif non_linearity == 'softplus':
        self.non_linearity = tf.keras.layers.Activation('softplus')
    else:
        self.non_linearity = tf.keras.layers.Activation(non_linearity)

def build(self, input_shape):
    if len(input_shape) > 0:
        self.fc3 = tf.keras.layers.Dense(input_shape[-1])
        self.built = True

@tf.function
def call(self, t, x, training=None, **kwargs):
    """
```

```
        Forward pass. If time dependent, concatenates the time
        dimension onto the input before the call to the dense layer.
        # Arguments:
            t: Tensor. Current time. Shape (1,).
            x: Tensor. Shape (batch_size, input_dim).
        # Returns:
            Output tensor of forward pass.
        """

        # build the final layer if it wasnt built yet
        if self.fc3 is None:
            self.fc3 = tf.keras.layers.Dense(x.shape.as_list()[-1])

        # Forward pass of model corresponds to one function evaluation, so
        # increment counter

        self.nfe += 1
        if self.time_dependent:
            # Shape (batch_size, 1)
            t_vec = tf.ones([x.shape[0], 1], dtype=t.dtype) * t
            # Shape (batch_size, data_dim + 1)
            t_and_x = tf.concat([t_vec, x], axis=-1)
            # Shape (batch_size, hidden_dim)
            # TODO: Remove cast when Keras supports double
            out = self.fc1(tf.cast(t_and_x, tf.float32))
        else:
            out = self.fc1(x)
        out = self.non_linearity(out)
        out = self.fc2(out)
        out = self.non_linearity(out)
        out = self.fc3(out)
        return out

class LatentODEFunc(tf.keras.models.Model):

    def __init__(self, input_dim, latent_dim, ode_func_net):
        super().__init__()
        self.input_dim = input_dim
        self.gradient_net = ode_func_net
```

```
    @tf.function
    def call(self, t_local, y, backwards=False):
        """
        Perform one step in solving ODE. Given current data point y and current time point t_local, returns gradient dy/dt at this time point
        t_local: current time point
        y: value at the current time point
        """
        #to avoid type incompatibility
        t_local, y = tf.cast(t_local, tf.float32), tf.cast(y, tf.float32)
        grad = self.get_ode_gradient_nn(t_local, y)
        if backwards:
            grad = -grad
        return grad

    def get_ode_gradient_nn(self, t_local, y):
        return self.gradient_net(y)

    def sample_next_point_from_prior(self, t_local, y):
        """
        t_local: current time point
        y: value at the current time point
        """
        #to avoid type incompatibility
        t_local, y = tf.cast(t_local, tf.float32), tf.cast(y, tf.float32)
        return self.get_ode_gradient_nn(t_local, y)

class LatentODEFuncPoisson(tf.keras.models.Model):

    def __init__(self, input_dim, latent_dim, ode_func_net, lambda_net):
        super().__init__()
        self.input_dim = input_dim
        self.latent_dim = latent_dim
        self.latent_ode = LatentODEFunc(input_dim = input_dim,
                                        latent_dim = latent_dim,
                                        ode_func_net = ode_func_net)
        #self.gradient_net = ode_func_net
        self.lambda_net = lambda_net
        # The computation of poisson likelihood can become numerically unstable.
        #The integral lambda(t) dt can take large values. In fact, it is equal to the expected number of events on the interval [0,T]
```

```
        #Exponent of lambda can also take large values
        #So we divide lambda by the constant and then multiply the integral of
lambda by the constant
        self.const_for_lambda = tf.constant([100.], tf.float32)

    #@tf.function
    def call(self, t_local, y, backwards=False):
        """
        Perform one step in solving ODE. Given current data point y and current time
point t_local, returns gradient dy/dt at this time point
        t_local: current time point
        y: value at the current time point
        """
        #to avoid type incompatibility
        t_local, y = tf.cast(t_local, tf.float32), tf.cast(y, tf.float32)

        grad = self.get_ode_gradient_nn(t_local, y)
        if backwards:
            grad = -grad
        return grad

    def get_ode_gradient_nn(self, t_local, y):
        y, log_lam, int_lambda, y_latent_lam = self.extract_poisson_rate(y,
final_result = False)
        dydt_dldt = self.latent_ode(t_local, y_latent_lam)

        log_lam = log_lam - tf.math.log(self.const_for_lambda)
        return tf.concat([dydt_dldt, tf.math.exp(log_lam)],-1)

    def extract_poisson_rate(self, augmented, final_result = True):
        y, log_lambdas, int_lambda = None, None, None

        assert(augmented.shape[-1] == self.latent_dim + self.input_dim)
        latent_lam_dim = self.latent_dim // 2

        if len(augmented.shape) == 3:
            int_lambda  = augmented[:,:,-self.input_dim:]
            y_latent_lam = augmented[:,:,:-self.input_dim]

            log_lambdas  = self.lambda_net(y_latent_lam[:,:,-latent_lam_dim:])
            y = y_latent_lam[:,:,:-latent_lam_dim]
```

```
        elif len(augmented.shape) == 4:
            int_lambda  = augmented[:,:,:,-self.input_dim:]
            y_latent_lam = augmented[:,:,:,:-self.input_dim]

            log_lambdas  = self.lambda_net(y_latent_lam[:,:,:,-latent_lam_dim:])
            y = y_latent_lam[:,:,:,:-latent_lam_dim]

        # Multiply the intergral over lambda by a constant
        # only when we have finished the integral computation (i.e. this is not a call in
get_ode_gradient_nn)
        if final_result:
            int_lambda = int_lambda * self.const_for_lambda

        # Latents for performing reconstruction (y) have the same size as latent
poisson rate (log_lambdas)
        assert(y.shape[-1] == latent_lam_dim)

        return y, log_lambdas, int_lambda, y_latent_lam

class ODEBlock(tf.keras.models.Model):

    def __init__(self, odefunc, is_conv=False, tol=1e-3, adjoint=False,
                 solver='dopri5', **kwargs):
        """
        Solves ODE defined by odefunc.
        # Arguments:
            odefunc : ODEFunc instance or Conv2dODEFunc instance
                Function defining dynamics of system.
            is_conv : bool
                If True, treats odefunc as a convolutional model.
            tol : float
                Error tolerance.
            adjoint : bool
                If True calculates gradient with adjoint solver, otherwise
                backpropagates directly through operations of ODE solver.
            solver: ODE solver. Defaults to DOPRI5.
        """
        dynamic = kwargs.pop('dynamic', True)
        super().__init__(**kwargs, dynamic=dynamic)
```

```
        self.adjoint = adjoint
        self.is_conv = is_conv
        self.odefunc = odefunc
        self.tol = tol
        self.method = solver
        self.channel_axis = 1 if tf.keras.backend.image_data_format() ==
'channels_first' else -1

        if solver == "dopri5":
            self.options = {'max_num_steps': MAX_NUM_STEPS}
        else:
            self.options = None

    @tf.function
    def call(self, x, training=None, eval_times=None, **kwargs):
        """
        Solves ODE starting from x.
        # Arguments:
            x: Tensor. Shape (batch_size, self.odefunc.data_dim)
            eval_times: None or tf.Tensor.
                If None, returns solution of ODE at final time t=1. If tf.Tensor
                then returns full ODE trajectory evaluated at points in eval_times.
        # Returns:
            Output tensor of forward pass.
        """

        # Forward pass corresponds to solving ODE, so reset number of function
        # evaluations counter

        self.odefunc.nfe = 0

        if eval_times is None:
            integration_time = tf.convert_to_tensor([0, 1], dtype=x.dtype)
        else:
            integration_time = tf.cast(eval_times, x.dtype)

        if self.odefunc.augment_dim > 0:
            if self.is_conv:
                # Add augmentation
                if self.channel_axis == 1:
                    batch_size, _, height, width = x.shape
```

```
            aug = tf.zeros([batch_size, self.odefunc.augment_dim,
                            height, width], dtype=x.dtype)

        else:
            batch_size, height, width, _ = x.shape

            aug = tf.zeros([batch_size, height, width,
                            self.odefunc.augment_dim], dtype=x.dtype)

        # Shape (batch_size, channels + augment_dim, height, width)
        x_aug = tf.concat([x, aug], axis=self.channel_axis)
      else:
        # Add augmentation
        aug = tf.zeros([x.shape[0], self.odefunc.augment_dim], dtype=x.dtype)
        # Shape (batch_size, data_dim + augment_dim)
        x_aug = tf.concat([x, aug], axis=-1)
    else:
      x_aug = x

    if self.adjoint:
      out = odeint_adjoint(self.odefunc, x_aug, integration_time,
                           rtol=self.tol, atol=self.tol, method=self.method,
                           options=self.options)
    else:
      out = odeint(self.odefunc, x_aug, integration_time,
                   rtol=self.tol, atol=self.tol, method=self.method,
                   options=self.options)

    if eval_times is None:
      return out[1]  # Return only final time
    else:
      return out

  def trajectory(self, x, timesteps):
    """Returns ODE trajectory.
    Parameters
    ----------
    x : torch.Tensor
      Shape (batch_size, self.odefunc.data_dim)
    timesteps : int
```

```
        Number of timesteps in trajectory.
    """
    integration_time = tf.linspace(0., 1., timesteps)
    return self.call(x, eval_times=integration_time)


class ODENet(tf.keras.models.Model):

  def __init__(self, hidden_dim, output_dim,
               augment_dim=0, time_dependent=False, non_linearity='relu',
               tol=1e-3, adjoint=False, solver='dopri5', **kwargs):
    """
    An ODEBlock followed by a Linear layer.
    # Arguments:
        hidden_dim : int
            Dimension of hidden layers.
        output_dim : int
            Dimension of output after hidden layer. Should be 1 for regression or
            num_classes for classification.
        augment_dim: int
            Dimension of augmentation. If 0 does not augment ODE, otherwise
augments
            it with augment_dim dimensions.
        time_dependent : bool
            If True adds time as input, making ODE time dependent.
        non_linearity : string
            One of 'relu' and 'softplus'
        tol : float
            Error tolerance.
        adjoint : bool
            If True calculates gradient with adjoint method, otherwise
            backpropagates directly through operations of ODE solver.
        solver: ODE solver. Defaults to DOPRI5.
    """
    dynamic = kwargs.pop('dynamic', True)
    super().__init__(**kwargs, dynamic=dynamic)

    self.hidden_dim = hidden_dim
    self.augment_dim = augment_dim
    self.output_dim = output_dim
    self.time_dependent = time_dependent
```

```
        self.tol = tol

        odefunc = ODEFunc(hidden_dim, augment_dim,
                          time_dependent, non_linearity)

        self.odeblock = ODEBlock(odefunc, tol=tol, adjoint=adjoint, solver=solver)
        self.linear_layer = tf.keras.layers.Dense(self.output_dim)

    #@tf.function
    def call(self, x, training=None, return_features=False):
        features = self.odeblock(x, training=training)

        # Remove cast when keras supports double
        pred = self.linear_layer(tf.cast(features, tf.float32))
        if return_features:
            return features, pred
        return pred

class Conv2dTime(tf.keras.models.Model):
    """
    Implements time dependent 2d convolutions, by appending the time variable as
    an extra channel.
    """
    def __init__(self, dim_out, kernel_size=3, stride=1, padding="valid", dilation=1,
                 bias=True, transpose=False):
        super().__init__()
        module = tf.keras.layers.Conv2DTranspose if transpose else
tf.keras.layers.Conv2D

        self._padding = padding
        self._layer = module(
            dim_out, kernel_size=(kernel_size, kernel_size), strides=(stride, stride),
padding=self._padding,
            dilation_rate=dilation,
            use_bias=bias
        )

        self.channel_axis = 1 if tf.keras.backend.image_data_format() ==
'channels_first' else -1

    #@tf.function
```

```
    def call(self, t, x, training=None, **kwargs):
        # Remove cast when Keras supports double
        t = tf.cast(t, x.dtype)

        if self.channel_axis == 1:
            # Shape (batch_size, 1, height, width)
            tt = tf.ones_like(x[:, :1, :, :], dtype=t.dtype) * t  # channel dim = 1

        else:
            # Shape (batch_size, height, width, 1)
            tt = tf.ones_like(x[:, :, :, :1], dtype=t.dtype) * t  # channel dim = -1

        ttx = tf.concat([tt, x], axis=self.channel_axis)  # concat at channel dim

        # Remove cast when Keras supports double
        ttx = tf.cast(ttx, tf.float32)
        return self._layer(ttx)


class Conv2dODEFunc(tf.keras.models.Model):

    def __init__(self, num_filters, augment_dim=0,
                 time_dependent=False, non_linearity='relu', **kwargs):
        """
        Convolutional block modeling the derivative of ODE system.
        # Arguments:
            num_filters : int
                Number of convolutional filters.
            augment_dim: int
                Number of augmentation channels to add. If 0 does not augment ODE.
            time_dependent : bool
                If True adds time as input, making ODE time dependent.
            non_linearity : string
                One of 'relu' and 'softplus'
        """
        dynamic = kwargs.pop('dynamic', True)
        super().__init__(**kwargs, dynamic=dynamic)

        self.augment_dim = augment_dim
        self.time_dependent = time_dependent
        self.nfe = 0  # Number of function evaluations
```

```
        # self.channels += augment_dim
        self.num_filters = num_filters

        if time_dependent:
            self.conv1 = Conv2dTime(self.num_filters,
                                    kernel_size=1, stride=1, padding=0)
            self.conv2 = Conv2dTime(self.num_filters,
                                    kernel_size=3, stride=1, padding=1)
            self.conv3 = None

        else:
            self.conv1 = tf.keras.layers.Conv2D(self.num_filters,
                                                kernel_size=(1, 1), strides=(1, 1),
                                                padding='valid')
            self.conv2 = tf.keras.layers.Conv2D(self.num_filters,
                                                kernel_size=(3, 3), strides=(1, 1),
                                                padding='same')
            self.conv3 = None

        if non_linearity == 'relu':
            self.non_linearity = tf.keras.layers.ReLU()
        elif non_linearity == 'softplus':
            self.non_linearity = tf.keras.layers.Activation('softplus')
        else:
            self.non_linearity = tf.keras.layers.Activation(non_linearity)

    def build(self, input_shape):
        if len(input_shape) > 0:
            if self.time_dependent:
                self.conv3 = Conv2dTime(self.channels,
                                        kernel_size=1, stride=1, padding=0)
            else:
                self.conv3 = tf.keras.layers.Conv2D(self.channels,
                                                    kernel_size=(1, 1), strides=(1, 1),
                                                    padding='valid')

            self.built = True

    #@tf.function
    def call(self, t, x, training=None, **kwargs):
        """
```

```
        Parameters
        ----------
        t : Tensor
            Current time.
        x : Tensor
            Shape (batch_size, input_dim)
        """
        # build the final layer if it wasnt built yet
        if self.conv3 is None:
            channel_dim = 1 if tf.keras.backend.image_data_format() == 'channel_first'
else -1
            self.channels = x.shape.as_list()[channel_dim]

            if self.time_dependent:
                self.conv3 = Conv2dTime(self.channels,
                                        kernel_size=1, stride=1, padding=0)
            else:
                self.conv3 = tf.keras.layers.Conv2D(self.channels,
                                                    kernel_size=(1, 1), strides=(1, 1),
                                                    padding='valid')

        self.nfe += 1

        if self.time_dependent:
            out = self.conv1(t, x)
            out = self.non_linearity(out)
            out = self.conv2(t, out)
            out = self.non_linearity(out)
            out = self.conv3(t, out)
        else:
            # TODO: Remove cast to tf.float32 once Keras supports tf.float64
            x = tf.cast(x, tf.float32)
            out = self.conv1(x)
            out = self.non_linearity(out)
            out = self.conv2(out)
            out = self.non_linearity(out)
            out = self.conv3(out)

        return out
```

```
class Conv1dTime(tf.keras.models.Model):
    """
    Implements time dependent 1d convolutions, by appending the time variable as
    an extra channel.
    """
    def __init__(self, dim_out, kernel_size=3, stride=1, padding="valid", dilation=1,
                 bias=True, data_format="channels_last"):
        super().__init__()
        module = tf.keras.layers.Conv1D

        self.data_format = data_format
        self._padding = padding
        self._layer = module(
            dim_out, kernel_size=kernel_size, strides=stride, padding=self._padding,
            dilation_rate=dilation,
            use_bias=bias, data_format=self.data_format
        )

        self.channel_axis = 1 if self.data_format == 'channels_first' else -1

    #@tf.function
    def call(self, t, x, training=None, **kwargs):
        # Remove cast when Keras supports double
        t = tf.cast(t, x.dtype)

        if self.channel_axis == 1:
            # Shape (batch_size, 1, length)
            tt = tf.ones_like(x[:, :1, :], dtype=t.dtype) * t  # channel dim = 1

        else:
            # Shape (batch_size, length, 1)
            tt = tf.ones_like(x[:, :, :1], dtype=t.dtype) * t  # channel dim = -1

        ttx = tf.concat([tt, x], axis=self.channel_axis)  # concat at channel dim

        # Remove cast when Keras supports double
        ttx = tf.cast(ttx, tf.float32)
        return self._layer(ttx)


class Conv1dODEFunc(tf.keras.models.Model):
```

```
   def __init__(self, num_filters, augment_dim=0,
                time_dependent=False, non_linearity='relu',
data_format="channels_last", **kwargs):
        """
        Convolutional block modeling the derivative of ODE system.
        # Arguments:
            num_filters : int
                Number of convolutional filters.
            augment_dim: int
                Number of augmentation channels to add. If 0 does not augment ODE.
            time_dependent : bool
                If True adds time as input, making ODE time dependent.
            non_linearity : string
                One of 'relu' and 'softplus'
        """
        dynamic = kwargs.pop('dynamic', True)
        super().__init__(**kwargs, dynamic=dynamic)

        self.augment_dim = augment_dim
        self.time_dependent = time_dependent
        self.nfe = 0  # Number of function evaluations
        # self.channels += augment_dim
        self.num_filters = num_filters
        self.data_format

        if time_dependent:
            self.conv1 = Conv1dTime(self.num_filters,
                                    kernel_size=1, stride=1, padding=0,
data_format=self.data_format)
            self.conv2 = Conv1dTime(self.num_filters,
                                    kernel_size=3, stride=1, padding=1,
data_format=self.data_format)
            self.conv3 = None

        else:
            self.conv1 = tf.keras.layers.Conv1D(self.num_filters,
                                        kernel_size=1, strides=1,
                                        padding='valid', data_format=self.data_format)
            self.conv2 = tf.keras.layers.Conv1D(self.num_filters,
                                        kernel_size=3, strides=1,
```

```
                                padding='same', data_format=self.data_format)
            self.conv3 = None

        if non_linearity == 'relu':
            self.non_linearity = tf.keras.layers.ReLU()
        elif non_linearity == 'softplus':
            self.non_linearity = tf.keras.layers.Activation('softplus')
        else:
            self.non_linearity = tf.keras.layers.Activation(non_linearity)

    def build(self, input_shape):
        if len(input_shape) > 0:
            if self.time_dependent:
                self.conv3 = Conv1dTime(self.channels,
                                        kernel_size=1, stride=1, padding=0,
data_format=self.data_format)
            else:
                self.conv3 = tf.keras.layers.Conv1D(self.channels,
                                            kernel_size=1, strides=1,
                                            padding='valid', data_format=self.data_format)

            self.built = True

    #@tf.function
    def call(self, t, x, training=None, **kwargs):
        """
        Parameters
        ----------
        t : Tensor
            Current time.
        x : Tensor
            Shape (batch_size, input_dim)
        """
        # build the final layer if it wasnt built yet
        if self.conv3 is None:
            channel_dim = 1 if self.data_format == 'channel_first' else -1
            self.channels = x.shape.as_list()[channel_dim]

            if self.time_dependent:
                self.conv3 = Conv1dTime(self.channels,
                                        kernel_size=1, stride=1, padding=0)
```

```
            else:
                self.conv3 = tf.keras.layers.Conv1D(self.channels,
                                                    kernel_size=(1, 1), strides=(1, 1),
                                                    padding='valid')

        self.nfe += 1

        if self.time_dependent:
            out = self.conv1(t, x)
            out = self.non_linearity(out)
            out = self.conv2(t, out)
            out = self.non_linearity(out)
            out = self.conv3(t, out)
        else:
            # TODO: Remove cast to tf.float32 once Keras supports tf.float64
            x = tf.cast(x, tf.float32)
            out = self.conv1(x)
            out = self.non_linearity(out)
            out = self.conv2(out)
            out = self.non_linearity(out)
            out = self.conv3(out)

        return out


class Conv2dODENet(tf.keras.models.Model):
    """Creates an ODEBlock with a convolutional ODEFunc followed by a Linear
    layer.
    Parameters
    ----------
    img_size : tuple of ints
        Tuple of (channels, height, width).
    num_filters : int
        Number of convolutional filters.
    output_dim : int
        Dimension of output after hidden layer. Should be 1 for regression or
        num_classes for classification.
    augment_dim: int
        Number of augmentation channels to add. If 0 does not augment ODE.
    time_dependent : bool
        If True adds time as input, making ODE time dependent.
```

```
non_linearity : string
    One of 'relu' and 'softplus'
tol : float
    Error tolerance.
adjoint : bool
    If True calculates gradient with adjoint method, otherwise
    backpropagates directly through operations of ODE solver.
return_sequences : bool
    Whether to return the Convolution outputs, or the features after an
    affine transform.
solver: ODE solver. Defaults to DOPRI5.
"""
def __init__(self, num_filters, output_dim=1,
             augment_dim=0, time_dependent=False, out_kernel_size=(1, 1),
             non_linearity='relu', out_strides=(1, 1),
             tol=1e-3, adjoint=False, solver='dopri5', **kwargs):

    dynamic = kwargs.pop('dynamic', True)
    super().__init__(**kwargs, dynamic=dynamic)

    self.num_filters = num_filters
    self.augment_dim = augment_dim
    self.output_dim = output_dim
    self.time_dependent = time_dependent
    self.tol = tol
    self.solver = solver
    self.output_kernel = out_kernel_size
    self.output_strides = out_strides

    odefunc = Conv2dODEFunc(num_filters, augment_dim,
                            time_dependent, non_linearity)

    self.odeblock = ODEBlock(odefunc, is_conv=True, tol=tol,
                             adjoint=adjoint, solver=solver)

    self.output_layer = tf.keras.layers.Conv2D(self.output_dim,
                                               kernel_size=out_kernel_size,
                                               strides=out_strides,
                                               padding='same')

#@tf.function
```

```
    def call(self, x, training=None, return_features=False):
        features = self.odeblock(x, training=training)

        # TODO: Remove cast when Keras supports double
        pred = self.output_layer(tf.cast(features, tf.float32))

        if return_features:
            return features, pred
        else:
            return pred


class Conv1dODENet(tf.keras.models.Model):
    """Creates an ODEBlock with a convolutional ODEFunc followed by a Linear
    layer.
    Parameters
    ----------
    img_size : tuple of ints
        Tuple of (sequence, channels, length).
    num_filters : int
        Number of convolutional filters.
    output_dim : int
        Dimension of output after hidden layer. Should be 1 for regression or
        num_classes for classification.
    augment_dim: int
        Number of augmentation channels to add. If 0 does not augment ODE.
    time_dependent : bool
        If True adds time as input, making ODE time dependent.
    non_linearity : string
        One of 'relu' and 'softplus'
    tol : float
        Error tolerance.
    adjoint : bool
        If True calculates gradient with adjoint method, otherwise
        backpropagates directly through operations of ODE solver.
    return_sequences : bool
        Whether to return the Convolution outputs, or the features after an
        affine transform.
    solver: ODE solver. Defaults to DOPRI5.
    """
    def __init__(self, num_filters, output_dim=1,
```

```
            augment_dim=0, time_dependent=False, out_kernel_size=1,
            non_linearity='relu', out_strides=1,
            tol=1e-3, adjoint=False, solver='dopri5', **kwargs):

        dynamic = kwargs.pop('dynamic', True)
        super().__init__(**kwargs, dynamic=dynamic)

        self.num_filters = num_filters
        self.augment_dim = augment_dim
        self.output_dim = output_dim
        self.time_dependent = time_dependent
        self.tol = tol
        self.solver = solver
        self.output_kernel = out_kernel_size
        self.output_strides = out_strides

        odefunc = Conv1dODEFunc(num_filters, augment_dim,
                                time_dependent, non_linearity)

        self.odeblock = ODEBlock(odefunc, is_conv=True, tol=tol,
                                 adjoint=adjoint, solver=solver)

        self.output_layer = tf.keras.layers.Conv1D(self.output_dim,
                                                   kernel_size=out_kernel_size,
                                                   strides=out_strides,
                                                   padding='same')

    #@tf.function
    def call(self, x, training=None, return_features=False):
        features = self.odeblock(x, training=training)

        # TODO: Remove cast when Keras supports double
        pred = self.output_layer(tf.cast(features, tf.float32))

        if return_features:
            return features, pred
        else:
            return pred

class DiffeqSolver(tf.keras.models.Model):
    def __init__(self, input_dim, ode_func, method, latents,
```

```
            odeint_rtol = 1e-4, odeint_atol = 1e-5, **kwargs):
        dynamic = kwargs.pop('dynamic', True)
        super().__init__()

        self.method = method
        self.latents = latents
        self.input_dim = input_dim
        self.ode_func = ode_func

        self.odeint_rtol = odeint_rtol
        self.odeint_atol = odeint_atol

    #@tf.function
    def call(self, first_point, time_steps_to_predict,
        backwards = False, adjoint=False):
        """
        Decode trajectory through an ODE Solver
        """
        first_point, time_steps_to_predict = tf.cast(first_point, tf.float32),
tf.cast(time_steps_to_predict, tf.float32)
        n_traj_samples, n_traj = first_point.shape[0], first_point.shape[1]
        n_dims = first_point.shape[-1]
        if adjoint:
            pred_y = odeint_adjoint(self.ode_func, first_point,time_steps_to_predict,
            rtol=self.odeint_rtol, atol=self.odeint_atol, method = self.method)
        else:
            pred_y = odeint(self.ode_func, first_point, time_steps_to_predict,
            rtol=self.odeint_rtol, atol=self.odeint_atol, method = self.method)
        #print('bef {}'.format(pred_y.shape))
        # shape: [n_traj_samples, n_traj, n_tp, n_dim]
        pred_y = tf.transpose(pred_y, perm=[1,2,0,3])
        #print('af {}'.format(pred_y.shape))
        return pred_y

    def sample_traj_from_prior(self, starting_point_enc, time_steps_to_predict,
        n_traj_samples = 1):
        """
        Decode the trajectory through ODE Solver using samples from the prior
        time_steps_to_predict: time steps at which we want to sample the new
trajectory
        """
```

```
        func = self.ode_func.sample_next_point_from_prior

        pred_y = odeint(func, starting_point_enc, time_steps_to_predict,
          rtol=self.odeint_rtol, atol=self.odeint_atol, method = self.method)
        # shape: [n_traj_samples, n_traj, n_tp, n_dim]
        pred_y = tf.transpose(pred_y, perm = [1,2,0,3])
        return pred_y

class LatentODEVAE(VAEBaseline):
    """
    Variational Autoencoder with Latent ODE dynamics modeller
    """
    def __init__(self,
                 input_dim,
                 latent_dim,
                 z0_prior,
                 encoder_z0,
                 decoder,
                 diffeq_solver,
                 obsrv_std=0.01,
                 use_binary_classif=False,
                 classif_per_tp=False,
                 use_poisson_proc=False,
                 linear_classifier=False,
                 n_labels=1,
                 train_classif_w_reconstr=False,
                 dynamic = True):
        super().__init__(input_dim, latent_dim, z0_prior)
        self.obsrv_std=obsrv_std
        self.use_binary_classif=use_binary_classif
        self.classif_per_tp=classif_per_tp
        self.use_poisson_proc=use_poisson_proc
        self.linear_classifier=linear_classifier
        self.n_labels=n_labels
        self.train_classif_w_reconstr=train_classif_w_reconstr
        self.encoder_z0 = encoder_z0
        self.diffeq_solver = diffeq_solver
        self.decoder = decoder
        self.use_poisson_proc = use_poisson_proc
        #self.dynamic = dynamic
```

```
    def call(self, data):
        n_traj_samples=1
        mode=None
        time_steps_to_predict, truth, truth_time_steps = data
        return self.get_reconstruction(time_steps_to_predict, truth, truth_time_steps,
n_traj_samples=1, mode=None)[0]

    def get_reconstruction(self, time_steps_to_predict, truth, truth_time_steps,
                           n_traj_samples=1, mode=None):

        if isinstance(self.encoder_z0, ODERNNEncoder):
            first_point_mu, first_point_std = self.encoder_z0(
                                        truth, truth_time_steps)
            means_z0 = tf.tile(first_point_mu, [n_traj_samples, 1, 1])
            sigma_z0 = tf.tile(first_point_std, [n_traj_samples, 1, 1])
            #print(means_z0)
            #print(sigma_z0)
            first_point_enc = tfp.distributions.Normal(loc=means_z0,
scale=sigma_z0).sample()
        else:
            raise Exception('Unlnown encoder type:
{}'.format(type(self.encoder_z0.__name__)))

        #first_point_enc = tf.abs(first_point_enc)

        if self.use_poisson_proc:
            n_traj_samples, n_traj, n_dims = first_point_enc.shape
            zeros = tf.zeros([n_traj_samples, n_traj,self.input_dim])
            first_point_enc_aug = tf.concat([first_point_enc, zeros], -1)
            means_z0_aug = tf.concat([means_z0, zeros],-1)
        else:
            first_point_enc_aug = first_point_enc
            means_z0_aug = means_z0

        # Shape of sol_y [n_traj_samples, n_samples, n_timepoints, n_latents]
        sol_y = self.diffeq_solver(first_point_enc_aug, time_steps_to_predict)



        if self.use_poisson_proc:
            sol_y, log_lambda_y, int_lambda, _ =
self.diffeq_solver.ode_func.extract_poisson_rate(sol_y)
```

```
        #print(sol_y)
        pred_x = self.decoder(sol_y)

        all_extra_info = {
            "first_point": (first_point_mu, first_point_std, first_point_enc),
            "latent_traj": tf.stop_gradient(sol_y)
        }

        if self.use_poisson_proc:
            # intergral of lambda from the last step of ODE Solver
            all_extra_info["int_lambda"] = int_lambda[:,:,-1,:]
            all_extra_info["log_lambda_y"] = log_lambda_y

        if self.use_binary_classif:
            if self.classif_per_tp:
                all_extra_info["label_predictions"] = self.classifier(sol_y)
            else:
                all_extra_info["label_predictions"] =
tf.squeeze(self.classifier(first_point_enc), -1)

        return pred_x, all_extra_info

    def sample_traj_from_prior(self, time_steps_to_predict, n_traj_samples=1):
        # Sample z0 from prior
        #starting_point_enc = tf.squeeze(self.z0_prior.sample([n_traj_samples, 1,
self.latent_dim]), -1)
        starting_point_enc = self.z0_prior.sample([n_traj_samples, 1,
self.latent_dim])

        starting_point_enc_aug = starting_point_enc
        if self.use_poisson_proc:
            n_traj_samples, n_traj, n_dims = starting_point_enc.shape
            # append a vector of zeros to compute the integral of lambda
            zeros = tf.zeros([n_traj_samples, n_traj,self.input_dim], tf.float32)
            starting_point_enc_aug = tf.concat([starting_point_enc, zeros], -1)

        sol_y = self.diffeq_solver.sample_traj_from_prior(starting_point_enc_aug,
time_steps_to_predict,
            n_traj_samples = 3)
```

```
        if self.use_poisson_proc:
            sol_y, log_lambda_y, int_lambda, _ =
self.diffeq_solver.ode_func.extract_poisson_rate(sol_y)

        return self.decoder(sol_y)
```