Comparative Analysis of Neural ODEs with LSTMs

Prasuk Jain, Pratyush Sinha, Mohit Sharma, and Raghav Chawla
Department of Mathematical Sciences, Indian Institute of Technology (BHU), Varanasi, India
prasuk.jain.mat23@iitbhu.ac.in, pratyush.sinha.mat23@iitbhu.ac.in, Mohit.sharma.mat23@iitbhu.ac.in, Raghav.chawla.mat23@iitbhu.ac.in

Abstract

In this paper, we dive deep into Neural Ordinary Differential Equations (Neural ODEs), a contemporary technique for modeling continuous-time dynamic systems, and compare it with conventional models like Long Short-Term Memory (LSTM) networks, Recurrent Neural Networks (RNNs), and linear regression. We investigate the ability of Neural ODEs to generalize the temporal dependencies with fewer parameters and demonstrate their application to various tasks.

Index Terms

Neural ODE, LSTM, RNN, Linear Regression, Dynamic Systems, Continuous-time Modeling

I. Introduction

Neural ordinary differential equations (Neural ODEs) have sprung into prominence over traditional discrete models, all owing to the latest upsurge in time-series modeling and dynamic system analysis. With Neural ODEs, neural networks and differential equation solvers are integrated in a single powerful family of models that provide continuous representations of temporal data. This feature is particularly useful in applications that require fine control over time evolution or adaptability to irregularly sampled data.

In contrast, traditional models like LSTMs and RNNs operate in discrete time steps, making them appropriate for structured time-series data but inconsistent in handling events with ill-defined time intervals. Moreover, linear regression is simple to compute but lacks the descriptive power to fit complex temporal patterns. This paper aims to:

- Present the mathematical formulation of Neural ODEs and compare it to LSTMs, RNNs, and linear regression.
- Evaluate their capacity for generalization in Neural ODEs expressed in time-series patterns.
- Demonstrate they are the novel in achieving comparable or even better performance with fewer parameters and better temporal flexibility.

II. Mathematical Formulation

The fundamental equation governing Neural ODEs can be expressed as:

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), \theta), \tag{1}$$

where $\mathbf{h}(t)$ denotes the hidden state at time t, and f is a neural network parameterized by θ. This continuous formulation it is different from discrete-time models like LSTMs and RNNs, where the state update depends on time-step intervals.

A. Comparison with LSTMs, RNNs, and Linear Regression

In traditional RNNs, the hidden state $\mathbf{h}_{t+1}$ at the next time step is calculated as:

$$\mathbf{h}_{t+1} = \sigma(\mathbf{W}_h\mathbf{h}_t + \mathbf{W}_x\mathbf{x}_t + \mathbf{b}), \tag{2}$$

where σ is the activation function, and $\mathbf{W}_h$, $\mathbf{W}_x$, and $\mathbf{b}$ are learnable parameters.

In LSTM networks the rules of updating are more complex since they contain gates that frame the flow of information. However, simple regression models model relationships using a weighted sum of variables such as:

$$y = \mathbf{w}^T\mathbf{x} + b, \tag{3}$$

which limits their capacity for sequential dependence.

III. Results

To conclude this section, the performance results of the models on various tasks will be briefly summarized here. Evaluations in terms of generalization capability, parameter efficiency, and temporal flexibility will all form part of the summaries.

CODE EXPLANATION

1. Data Import and Initial Setup

The code begins by including all the necessary Julia packages to handle data processing, modeling, and visualization tasks. Packages such as `Optimization`, `Lux`, and `DiffEqFlux` are helpful with machine learning and differential equations modeling, and `DataFrames` and `CSV` are used for data manipulation and reading CSV files. `Plots` included for making visualizations of data.

2. Loading the Dataset

The data is read into a DataFrame from a CSV file. Specific data types are assigned to each column for correct parsing of data. The columns contain details about date, temperature, humidity, wind speed, sea level pressure, and moon phase. This makes each column processable depending on the data type, based on which further analysis can be carried out.

3. Feature Configuration

Arrays define the key features, units, and names of the features. This setup organizes how the dataset columns by correlating the feature labels with a unit and provides meaningful identifiers for each data attribute (e.g., temperature, humidity, etc.).

4. Handling Missing Data

The code checks each column for missing values and prints the count. It then drops any rows that contain a missing value and makes sure to print the count of missing values after this recalculation. The aim here is to make sure the dataset used in analysis is devoid of missing data points, as missing information can cause a big issue in reliable modeling.

5. Normalization Function

A normalization function is defined to scale the dataset's values to have zero mean and unit variance. This function returns normalized values with their mean and standard deviation to allow any future data to be scaled back to its original range. Normalization maximizes a machine learning model performance and improves the rate at which it converges.

6. Feature Plotting Function

A function is defined to create time series plots for each feature. It produces subplots, which, for every subplot, plots a feature (such as temperature or humidity) over time. This visualization allows one to find patterns and trends in each feature over the time span of the dataset.

7. Data Preprocessing

It extracts out of the date column year and month and converts them into numeric format; then, it groups the data by year and month and takes an average of each feature. It normalizes this data and prepares train and test sets from it, returning normalized data along with scaling factors, mean and standard deviation. This procedure prepares the data to model training and testing concerning structuring, normalizing, and splitting it up.

8. Additional Imports

Additional packages `ComponentArrays` and `BSON` are imported. `ComponentArrays` allows for handling complex data structures, and `BSON` provides utilities to save and load models and data, enabling persistence and reuse of trained models.

CODE EXPLANATION FOR NEURAL ODE FORMATION AND TRAINING

1. Residual Block Structure

A mutable struct named `ResidualBlock` is defined, which contains an `inner_layer` represented as a `Chain`. This struct is meant to encapsulate components of a residual block so that deep learning architectures with skip connections can be constructed.

2. Residual Block Constructor

A function `ResidualBlock(input_dim, hidden_dim)` is defined to create an instance of the `ResidualBlock`. Inside this function, a `Chain` is constructed with two dense layers:

- The first layer maps the input dimension to a hidden dimension using the `swish` activation function.
- The second layer maps the hidden dimension back to the original input dimension.

This architecture enables learning residual representations which improves convergence while training deep networks.

3. Neural ODE Function

The function `neural_ode(t, data_dim)` defines a Neural ODE model. It creates a neural network `f` using a sequence of dense layers with `swish` activation functions:

- The first layer transforms the input dimension to 32.
- The second layer transforms it to 64.
- The third layer maintains the 64 units.
- The fourth layer reduces it back to 32.
- The final layer maps it back to the original data dimension.

Next, a `NeuralODE` instance is created using the defined neural network, along with specific settings such as time extrema, the Tsit5 solver, save points at each time step, and absolute and relative tolerances for numerical integration.

The function also initializes a random number generator, sets up the neural network parameters `p`, and initializes the model state `state`.

4. Training One Round Function

The `train_one_round` function executes one round of training for the Neural ODE model. It takes several parameters, including the neural ODE instance `node`, model parameters , state, observed data `y`, optimizer settings `opt`, maximum iterations `maxiters`, and a random number generator `rng`. The function contains:

- A nested function `predict()` that generates predictions from the neural ODE model.
- A loss function `loss()` that calculates the mean squared error between the predicted and actual values.

It uses the `Optimization.AutoZygote()` for enabling automatic differentiation, sets up an optimization problem, and solves it using the Adam optimizer for a preset number of iterations. It returns the optimized parameters and the state updated by those parameters.

5. Training Function

The `train` function manages the training process across multiple observation points defined in `obs_grid`. It initializes empty arrays to store parameter histories `s` and loss values `losses`. For each observation in the grid:

- A new Neural ODE model is created with `neural_ode(t, size(y, 1))`.
- If the parameters or the state are uninitialized, they are set to the new values.
- The `train_one_round` function is called to train the model, updating the parameters and state.

The final loss and optimized parameters along with state and history of losses are gathered and returned by the function after iterating through the observation grid. It thus gives a complete view of the training process.

Conclusion

The code basically implements the architecture and training procedure of a Neural ODE, with residual blocks and dense layers to model continuous-time dynamics. The modular form allows easy experimentation with various configurations and training strategies.

CODE EXPLANATION FOR NEURAL ODE TRAINING AND PREDICTION

1. Training Observation Grid

The variable `obs_grid` is declared to vary within the range from 4 to `length(t_train)` with a step size of 4, i.e., `obs_grid = 4:4:length(t_train)`. This spec- ifies that during the setup, the training will begin with the increasing number of observa- tions beginning from the fourth observation and proceed to adaptively train the Neural ODE model on growing size of datasets.

2. Training Parameters

Two very important parameters for the training process are defined:

- `maxiters = 400`: This has been used to set the maximum number of iterations for training in the optimization routine to ensure convergence but not overfitting.
- `lr = 5e-3`: This learning rate parameter specifies the extent to which the optimizer changes the model parameters in training. A learning rate of 5×10^{-3} will therefore be generally characterized as a rather aggressive learning rate.

3. Training the Neural ODE

The function `train` is called with several parameters: `t_train` (time training data), `y_train` (corresponding training data), `obs_grid` (defined observation grid), `maxiters` (maximum iterations), `lr` (learning rate), `rng` (random number generator), and `progress=true` (which likely indicates whether to display progress during training). The output consists of:

- `s`: The history of model parameters across different training rounds.
- `state`: The final state of the neural ODE model after training.
- `losses`: A record of loss values at each iteration, providing insight into the model's training performance.
- `final_loss`: The value of loss after the training process is finished serves as an indication of how well the model is fitted to the training data.

4. Prediction Function

The `predict` function is defined to make predictions using the trained Neural ODE model. It takes three parameters:

- `y0`: The initial state of the system for which predictions are to be made.
- `t`: The time points at which predictions should be made.
- : The model parameters obtained from training.
- `state`: The current state of the Neural ODE model.

Inside the function:

- A new Neural ODE model is instantiated by calling `neural_ode(t, length(y0))`.
- Predictions `y` are generated by calling the model with the initial state `y0`, parameters , and state, returns the predicted values as an array.

Adaptive training of a Neural ODE model is illustrated in this code snippet by increasing the number of observation points and thus offsetting the mechanism for prediction methods using that trained model. It is a clear-cut depiction of all three major phases, modeling, evaluation of the model, and prediction involving an essential feature for time-series forecasting tasks." article graphicx

RESULTS

Feature	LSTM Error	NODE Error
Temperature	26.6589	20.2607
Humidity	131.5260	103.9060
Windspeed	173.6247	120.5372
Sea Level Pressure	278.3550	175.0130
Moonphase	0.00056	1.6740

TABLE I
COMPARISON OF MSE ERRORS

The distinction between Neural ODE (NODE) and LSTM models is uncovered during the examination of the different performance features. For example, in Temperature, NODE is better than LSTM by 24%, because it achieves a lower error of 20.2607 rather than 26.6589. In the case of Humidity, NODE states a 21% development of the error of 103.9060 in comparison to LSTM's error of 131.5260. Likewise, in the case of Windspeed, NODE is superior to LSTM by 30%, decreasing the deviation to 120.5372 from 173.6247. The biggest difference can be found at the Sea Level Pressure: here NODE gains the most by 37%, with 175.0130 being the error experienced by NODE and 278.3550 for LSTM. However, LSTM is highly superior to NODE for Moonphase, judging by the infinitesimal error of 0.00056 while NODE has 1.6740 for its opponent. In fact, most features are the principal strengths of NODE, nevertheless, the impressive performance of LSTM is showcased in the scenario of Moonphase where the machine model shows a low error value.

CONCLUSION

As can be seen from the table, Neural ODE (NODE) continuously beats LSTM in the majority of the variables such as Temperature, Humidity, Windspeed, and Sea Level Pressure with the improvement of 21% to 37%. This performance difference can be drawn from the principle that NODE is constructed to model continuous-time dynamics, which gives it the opportunity to recognize even more sophisticated relationships in time-series data. Its capacity to transform continuously is what makes it capable of modeling the inner physical process in features with more and more complicated patterns, which in turn reduces the errors.

Nonetheless, LSTM is best for dealing with cycled data like Moonphase. The Moonphase feature has cyclic, repeating patterns for which the LSTM recurrent architecture is well-suited. The LSTM cyclical behavior capturing ability makes it

perform well significantly on this feature, as evidenced by its very low error compared to NODE's higher error. Hence, even though NODE is well-suited for features with complex dynamics, LSTM has a more meaningful use for cyclic data because of its inherent ability to model periodic patterns.