Polynomial Neural Layers for Numerical Modeling of Dynamical Processes

Ilya Klimenko, Anna Golovkina(✉), and Vladimir Ruzhnikov

Saint Petersburg State University, Saint Petersburg, Russia
a.golovkina@spbu.ru

Abstract. The paper aims to present a description of the tm-flow library for the "flowly" construction and training of polynomial neural networks (PNN) for time-evolving process prediction. The introduced polynomial models have a strong relationship to dynamic systems that can be described by a system of nonlinear differential equations. The paper provides implementation details and an explanation of training strategies, along with a few illustrative numerical examples. The source code is available at https://github.com/PNN-Lab/tmflow/.

Keywords: polynomial neural layers · dynamical processes modeling · neural networks training

1 Introduction

Recently, there has been a lot of focus on sophisticated system modeling methodologies. This is explained by the necessity to address problems related to nonlinear systems and high dimensionality while at the same time providing good accuracy and generalization capabilities. Neural networks take an important place among the different advanced techniques of system modeling. However, most of the existing architecture suffers from poor explainability of the outcomes and extrapolation properties.

Polynomial neural networks (PNNs) are a type of neural network that uses polynomial activation functions to model the non-linear relationships between inputs and outputs. Polynomials are universal approximators that can be used to model all possible feature interactions and provide a rubric for interpretability [4]. PNNs have shown promising results for time-series prediction [2], especially for data with non-linear relationships and complex temporal dependencies. Moreover, polynomial functions have been widely studied by mathematicians and have been shown to have some useful theoretical properties. Polynomial neural networks also link the investigation of generic neural structures to the investigation of polynomial function characteristics [11].

A Group Method of Data Handling (GMDH) is one of the earliest methods for the systematic construction of nonlinear connections with PNN. GMDH was

Supported by Saint Petersburg State University, project ID: 94029367.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
O. Gervasi et al. (Eds.): ICCSA 2023 Workshops, LNCS 14109, pp. 261–273, 2023.
https://doi.org/10.1007/978-3-031-37120-2_17

developed in the late 1960s by Ivakhnenko [7] to identify nonlinear relations between input and output variables. The GMDH algorithm generates an optimal structure for the model through successive generations of partial descriptions. In comparison to well-known neural networks whose topologies are commonly fixed prior to all detailed (parametric) learning, the GMDH architecture is not fixed in advance but becomes fully optimized (both structurally and parametrically). However, it has drawbacks such as complex polynomials for simple systems and overly complex networks for nonlinear systems.

Another enhancement of PNN can be drawn from [12] where the pi-sigma network is introduced, which is a neural network with a single hidden layer. Improvements include regularization and multiple product units. However, it does not scale well in high-dimensional signals, and its experimental evaluation is conducted only on signals with known ground-truth distributions.

This paper discusses the construction of the architecture and training scheme of a recurrent polynomial neural network for modeling dynamic processes that can be described by stationary or non-stationary systems of ordinary differential equations. This choice is explained by the ability of the Taylor mapping technique to express the solution of an ordinary differential equation (ODE) in the form of a polynomial neural network [5]. According to the deterministic algorithm described in [8], such a PNN can be unambiguously initialized if the ODE corresponding to the modelled process is known. However, even with good initialization, the weight matrices may need to be fine-tuned due to the following factors:

- The ODE can be recovered from the data, but not exactly due to the specifics of the recovery methods;
- ODE can approximately describe the simulated process;
- In real-world conditions, the process can be influenced by various external forces that are not taken into account in the mathematical model, for one reason or another.

During the training, various regularization techniques can be used.

The paper provides the implementation details of different polynomial models and layers as part of the open-sourced library `tm-flow` available at https://github.com/PNN-Lab/tmflow/.

The rest of the paper is organized as follows: Sect. 2 contains a detailed description of the polynomial layer architecture, differentiating between the basic polynomial implementation and Kroneker's layer. Section 3 introduces different types of polynomial models built based on the described layers. Section 4 shows the main strategies of training and possible enhancements with regularization, masking, and normalization, while Sect. 5 illustrates the proposed architectures in a numerical example that can be found at https://github.com/PNN-Lab/tmflow/tree/main/examples/learn.

2 Architecture of the Polynomial Neural Layer

In this section, we go through the algorithmic specifics and architecture of polynomial layers. Generally, a polynomial layer is defined by three parameters:

1. Dimension of the input vector (`input_dim`)
2. Dimension of the output vector (`output_dim`)
3. Order of nonlinearity (`order`)

The dimension of the input vector usually corresponds to the dimension of the phase vector of a system of ordinary differential equations, but it can also include exogenous variables that can act as control parameters. The dimension of the output vector is determined by the number of measured phase variables, i.e., those for which training data is available (most often, their number coincides with the dimension of the input vector). The order of nonlinearity can be chosen arbitrarily in accordance with the features of the problem being solved and probable information about the dependence's nature.

The polynomial layer belongs to the class of recurrent and transforms the input vector into the output according to the following rule:

$$\boldsymbol{X}\left(t_{i+1}\right) = \text{bias} + \sum_{j=1}^{N} R^j \boldsymbol{X}\left(t_i\right)^{[j]} \tag{1}$$

where $\boldsymbol{X}\left(t_i\right)$ are the values of the input vector measured at time t_i, $\boldsymbol{X}\left(t_{i+1}\right)$ are the predicted values of the output vector at time t_{i+1}, bias is an optional parameter, N is the order of nonlinearity, $[j]$ is the nonlinearity order used to transform the input vector, $R^j \quad (j = 1 \ldots N)$ are the weight matrices of the layer. There is a mechanism for initializing R^j with matrices based on the algorithm given in [8]. In the absence of initializing matrices, all nonlinear orders are first initialized with zeros, while the matrix of the linear order is assigned a unit matrix. Naturally, for complex systems, this situation can cause a considerable increase in network training time and a decline in the neural network model's prediction abilities.

TensorFlow (TF) [1] and Keras libraries for Python were used to develop the polynomial neural layer, which provides a wide variety of built-in tools that facilitate neural layer and model construction and training. Furthermore, TensorFlow allows to leverage a graphics processing unit (GPU) to boost the neural network's learning performance.

Figure 1 shows the classes inheritance implementing polynomial layers derived of `tensorflow.keras.layers.Layer`. `base_polynomial_layer` contains the basic features that are further redefined in two polynomial layers realizations: basic polynomial and Kronecker. The following subsections provide a more extensive description of their differences.

2.1 Polynomial Layer

The polynomial layer (`polynomial_layer`) is derived from `base_polynomial_layer` and overrides the methods `build` and `call` from `tensorflow.keras.layers.Layer`:

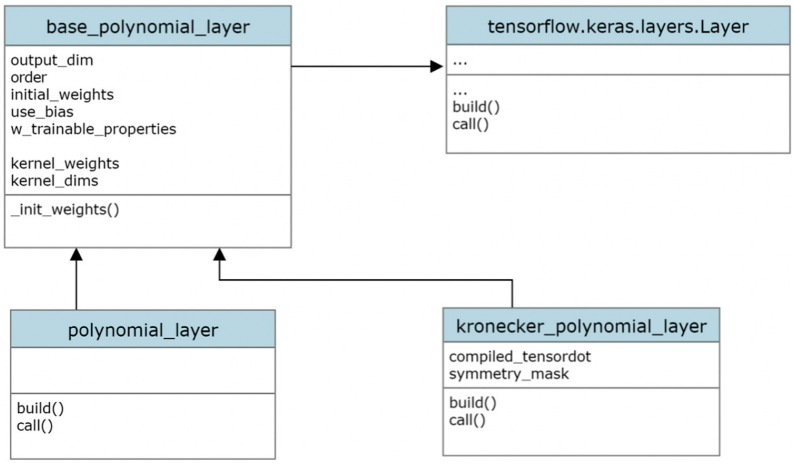

Fig. 1. Inheritance scheme of classes implementing polynomial layers

– (*build*) The dimensions of the weight matrices are computed during the layer construction step for each order of nonlinearity used in the layer. In case of basic polynomials, the dimension of the j-th matrix is $\left[n^j, m\right], j = 1 \ldots N$, where n is the dimension of the input vector, m is the dimension of the output vector. Number of rows in weight matrices relates to the number of combinations of the components of the input vector utilized to generate the j-th order of nonlinearity (repetition included). There is a possibility to initialize the Weight matrices before training.
– (*call*) According to the formula (1), in order to produce the output vector, the input space must be expanded using a polynomial nonlinear transformation of the input vector. This is carried out by calculating the recurrent tensor product with the formula

$$\boldsymbol{X}^k = \boldsymbol{X}^{k-1} \otimes \boldsymbol{X} \tag{2}$$

employing `einsum` function from the TensorFlow library, which applies the Einstein convention on summing the indexes of its arguments.
Thus, the procedure of the output vector calculation consists of two stages:

1. Initialization: $\overline{\mathbf{X}} = \text{bias}$;
2. In the cycle (j) from 1 to n:
   (a) Multiply the matrix by a vector and sum the result and $\overline{\mathbf{X}}$;
   (b) Raise the vector to the power of $j + 1$ using (2);

It is worth noticing that the polynomial layer implementation does not include any dimension reduction operations. For example, when we elevate an input vector of size 2 to the second power, we get:

$$\boldsymbol{Y} = R^2 \cdot \boldsymbol{X}^2 = \begin{pmatrix} r_{11}\ r_{12}\ r_{13}\ r_{14} \\ r_{21}\ r_{22}\ r_{23}\ r_{24} \\ r_{31}\ r_{32}\ r_{33}\ r_{34} \\ r_{41}\ r_{42}\ r_{43}\ r_{44} \end{pmatrix} \begin{pmatrix} x_1^2 \\ x_1 x_2 \\ x_2 x_1 \\ x_2^2 \end{pmatrix} \tag{3}$$

As seen in (3), when multiplying the matrix by a vector, the second and third components of the vector $\mathbf{Y}$ are the same, implying that the vector $\mathbf{Y}$'s dimension can be reduced by one. A similar scenario develops when using larger input vectors and higher degrees. A Kronecker layer is developed to actualize the potential of deleting repeating nonlinear components and lowering the dimension of the weight coefficient matrices. The main aspects of Kronecker layer are explained below.

2.2 Kronecker Layer

The main feature of `Kronecker_layer` layer is the implementation of Kronecker exponentiation operations associated with removing duplicate (up to the permutation of the multipliers) elements in the polynomialy transformed input vector. Similarly to `polynomial_layer` it redefines the abstract methods `build` and `call` of `tensorflow.keras.layers.Layer`.

- (*build*) The dimensions of the weight matrices are computed during the layer construction step for each order of nonlinearity used in the layer. In case of Kroneker's polynomials, the dimension of the j-th matrix is $\left[C_j^{n+j-1}, m\right], j = 1 \dots N$, where n is the dimension of the input vector, m is the dimension of the output vector, $C_j^{n+j-1} = \frac{(n+j-1)!}{j!(n-1)!}$. Number of rows in weight matrices relates to the number of different combinations of the components of the input vector (up to a permutation of factors) utilized to generate the j-th order of nonlinearity. In addition, a set of binary masks is calculated, which is used to eliminate identical components from the formula's exponentiation result (2).
- (*call*) Similarly to the `call` method in the polynomial layer, the input space is extended and duplicates are deleted by employing a polynomial nonlinear modification of the input vector. The following formula, for example, computes the second Kronecker degree for the given input vector of size two:

$$X^{[2]} = \boldsymbol{X}^2 \overset{*}{\odot} mask = \begin{pmatrix} x_1^2 \\ x_1 x_2 \\ x_2 x_1 \\ x_2^2 \end{pmatrix} \odot \begin{pmatrix} \text{True} \\ \text{True} \\ \text{False} \\ \text{True} \end{pmatrix} = \begin{pmatrix} x_1^2 \\ x_1 x_2 \\ x_2^2 \end{pmatrix}.$$

Here $\odot^*$ is the Hadamard product with a decrease in vector dimension (all vector elements with `False` in the appropriate position of the Boolean mask are eliminated). In this scenario, formula (3) will take the following form:

$$\boldsymbol{Y} = \hat{R}^2 \cdot \boldsymbol{X}^{[2]} = \begin{pmatrix} \hat{r}_{11}\ \hat{r}_{12}\ \hat{r}_{13} \\ \hat{r}_{21}\ \hat{r}_{22}\ \hat{r}_{23} \\ \hat{r}_{31}\ \hat{r}_{32}\ \hat{r}_{33} \end{pmatrix} \begin{pmatrix} x_1^2 \\ x_1 x_2 \\ x_2^2 \end{pmatrix}.$$

The matrix $\hat{R}^2$ varies from the matrix $R2$ in that the dimension is reduced.

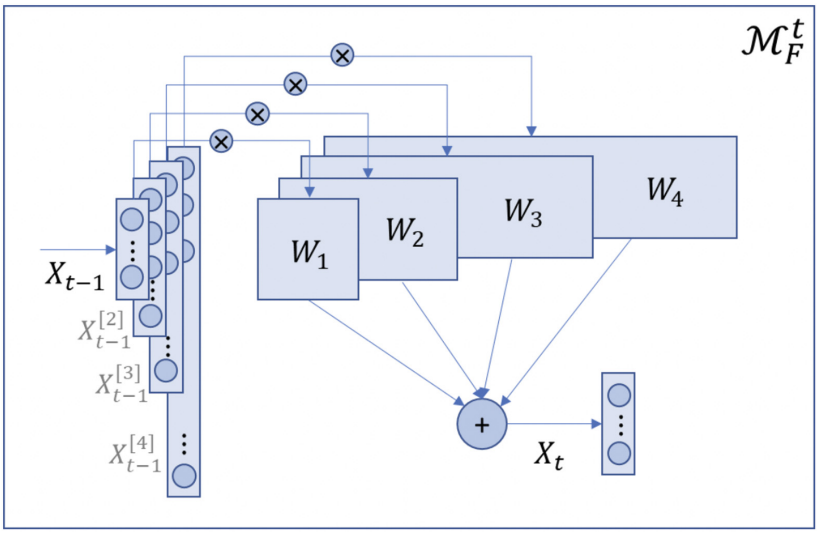

Fig. 2. The scheme of operation of the Kronecker layer in accordance with the formula (1) [5]

3 Polynomial Models

Models for a variety of problems may be created based on the types of polynomial layers covered above (with/without repeats of nonlinear components). The Kronecker layer is a priority building element for the polynomial models discussed below because it reduces the dimensionality of the weight coefficient matrices, reducing the number of free parameters for the optimizer and increasing the speed of output and training of the layer.

We consider two types of polynomial models suitable for describing stationary and non-stationary dynamical processes.

3.1 Polynomial Model with One Output (Single Output Model)

For both stationary and non-stationary systems, a model with a single output may be produced. Because the weight coefficient matrices in the stationary scenario depend only on the discreteness of the time series and not on a given instant in time, such a model has a single Kronecker layer (Fig. 2a)). In the case of a non-stationary system, one can try to bring it to a stationary form by injecting exogenous time-dependent variables into the phase vector, or one can design a multilayer model conforming, for example, to data periodicity (Fig. 2b)). The following procedure is used to train a model with a single output.

I Data Preparation

1. The data obtained as a result of measurement on k trajectories of the simulated dynamic system are given:

$$\boldsymbol{X}_1(t_1), \ldots, \boldsymbol{X}_1(t_{p_1}), \boldsymbol{X}_2(t_{p_1+1}), \ldots, \boldsymbol{X}_2(t_{p_1+p_2}), \ldots,$$

$$\boldsymbol{X}_k\left(t_{p_1+\ldots+p_{k-1}+1}\right), \ldots, \boldsymbol{X}_{\boldsymbol{k}}(t_{p_1+\ldots+p_k}),$$

where, $t_{p_1+\ldots+p_i+J} < t_{p_1+\ldots+p_i+J+1}, \forall j < p_{i+1}, p_i-$, is the number of vectors of the system state measured on the i-th trajectory with the initial vector $\boldsymbol{X}_i\left(t_{p_1+\ldots+p_{i-1}+1}\right)$.
2. Forming a set of training pairs $\{(\boldsymbol{X_{i+1}}(t_{p_1+\ldots+p_i+J}), \boldsymbol{X_{i+1}}(t_{p_1+\ldots+p_i+J+1})), \forall j < p_{i+1}, i = 1, k\}$, where the first vector of the pair is the training input vector, the second vector of the pair is the training output vector

The first vector in the training pair is denoted as $\boldsymbol{X}(t_0)$, and the second by $\boldsymbol{X}(t_M)$, where M is the number of Kronecker layers in the model (1 for a stationary system).

II Training

1. The vector $\boldsymbol{X}(t_0)$ is fed into the input of the Kronecker layer, and the vector $\boldsymbol{X}(t_M)$ is calculated;
2. The resultant vector $X(t_{\mathrm{M}})$ is fed into the input of the Kronecker layer, and the vector $X(t_{2\mathrm{M}})$ is calculated;
3. This process is repeated a specified number of times.

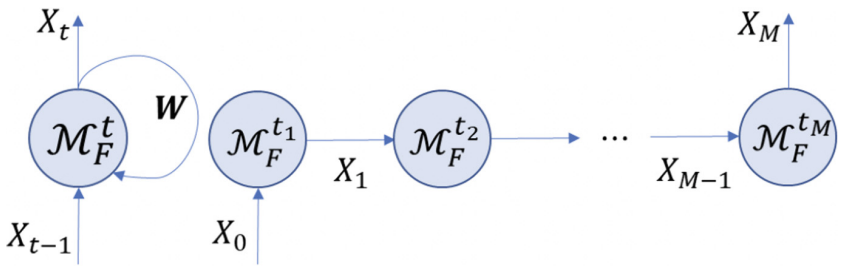

Fig. 3. Polynomial model with one output to describe a) stationary b) non-stationary dynamical process

3.2 Polynomial Model with Multiple Outputs (Multiple Output Model)

A polynomial model with multiple outputs is constructed out of a series of Kronecker layers. The weight matrices in a stationary case are fixed between layers and updated synchronously during the training. For a non-stationary model, the overall architecture does not change, but the output is formed from each hidden layer of the model (Fig. 3b)) [3].

The number of hidden Kronecker layers is usually dictated by the length of the training data (if it is supplied by a sequence of discrete trajectories with varied dynamic system parameters) or by the periodicity/seasonality of the time series used for training.

Figure 4 shows a model with multiple outputs for a stationary system. In this case, a vector $\boldsymbol{X}(t_0)$ is fed into the input, and an array of vectors is obtained at the output, $\boldsymbol{X}(t_1), \mathbf{X}(\mathrm{t}_2), \ldots, \mathbf{X}(\mathrm{t}_q)$. Accordingly, a set of training data is prepared for such a model.

I Data Preparation

1. The data obtained as a result of measurement on k trajectories of the simulated dynamic system are given:

$$\boldsymbol{X}_1(t_1), \ldots, \boldsymbol{X}_1(t_{p_1}), \boldsymbol{X}_2(t_{p_1+1}), \ldots, \boldsymbol{X}_2(t_{p_1+p_2}), \ldots,$$
$$\boldsymbol{X}_k\left(t_{p_1+\ldots+p_{k-1}+1}\right), \ldots, \boldsymbol{X}_{\boldsymbol{k}}(t_{p_1+\ldots+p_k}),$$

   Here, $t_{p_1+\ldots+p_i+J} < t_{p_1+\ldots+p_i+J+1}, \forall j < p_{i+1}, p_i -$, is the number of vectors of the system state measured on the i-th trajectory with the initial vector $\boldsymbol{X}_i\left(t_{p_1+\ldots+p_{i-1}+1}\right)$
2. Forming a set of training pairs

$$\{(\boldsymbol{X}_{i+1}(t_{p_1+\ldots}), [\boldsymbol{X}_{i+1}(t_{p_1+\ldots}), \ldots, \boldsymbol{X}_{i+1}(t_{p_1+\ldots})]), i = 0, k-1\},$$

   where the first vector of the pair is the training input vector, the second vector of the pair is a list containing a set of output vectors at each time.

The first vector in the training pair is denoted by $\boldsymbol{X}(t_0)$, the second by $[\boldsymbol{X}(t_1), \ldots, \boldsymbol{X}(t_q)]$, where q is the number of hidden Kronecker layers in the model (see Fig. 4 for an example).

II Training

1. The vector $\boldsymbol{X}(t_0)$ is fed into the input to the Kronecker layer, and a set of vectors is calculated $[\boldsymbol{X}(t_1), \ldots, \boldsymbol{X}(t_q)]$;
2. The resulting vector $\boldsymbol{X}(t_q)$ is fed into the input to the Kronecker layer, and the vector is calculated $[\boldsymbol{X}(t_{q+1}), \ldots, \boldsymbol{X}(t_{2q})]$;
3. This procedure is performed a certain number of times.

Because the computation of gradients takes into consideration the mismatch of outputs throughout the full length of the dynamic system's trajectory, rather than only between pairs of nearby locations, training a model with multiple outputs allows for dynamic consistency of predictions.

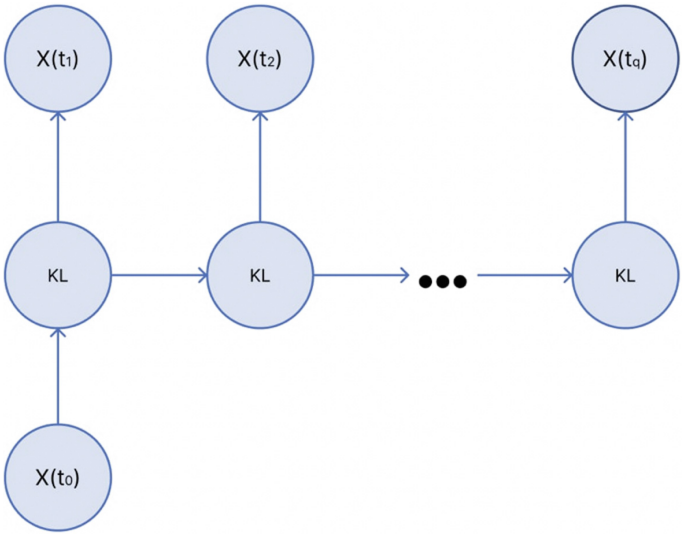

Fig. 4. Polynomial model with multiple outputs

4 Features of Training

This section discusses the possible options that can be employed for training polynomial models. Generally, training can be implemented in two ways:

1. *Training of all weight matrices at once.* This method implies training of all weight matrices simultaneously, i.e. the gradient is computed for all weight coefficients at the same time, and all weight matrices are altered at each step of training. The number of free coefficients can be enormous using this technique, complicating optimization algorithm convergence (particularly in the absence of competent initialization) and can potentially lead to physically incorrect predictions.
2. *Sequential training of matrices, starting from the bias and linear term matrix* R^1 *to the matrix* R^n *corresponding to the n-th order of nonlinearity.* Simultaneous weight matrix training ignores a crucial characteristic of the polynomial model: the contribution of weight coefficient matrices to the outcome (see formula (1)) is different. The linear component has the most effect on R^1; but, as the order of nonlinearity increases, the contribution decreases. In this regard, the sequential learning mechanism can be applied:
   - a linear model with a matrix $R1$ is being constructed and trained until the loss function stops decreasing;
   - a second-order model is being built, the linear part of which is initialized by a trained matrix R^1 loaded from the previous model. In this case, R^1 it is fixed, and only the matrix R^2 is trained, etc.

For models with a single output, a basic approach to learning is used, which assumes that a vector $\tilde{X}(t_0)$ is fed into the neural network's input, and a vector

$\tilde{X}(t_1)$ is calculated and compared to the reference vector $\overline{X}(t_0)$. The error function's value $\mathcal{L}$ is then determined, which can be i.e. MSE (mean square error), RMSE (root mean square error), or RMSLE (Root Mean Squared Log Error).

The gradient of the error function is then computed and utilized to alter the weight matrices. One of the primary benefits of this strategy is the rapid learning. One downside is that the placements of non-zero coefficients are not regulated throughout the learning process, however this may be remedied using various regularization approaches.

The following approach is used to train polynomial models with multiple outputs:

1. The input: vector $\tilde{\mathbf{X}}(t_0)$ and the reference values of the vectors $\bar{\mathbf{X}}(t_1)$, $\bar{\mathbf{X}}(t_2), \ldots, \bar{\mathbf{X}}(\mathrm{t}_q)$;
2. An array of vectors is calculated, $\tilde{\mathbf{X}}(t_1), \tilde{\mathbf{X}}(\mathrm{t}_2), \ldots, \tilde{\mathbf{X}}(\mathrm{t}_q)$;
3. For a pair of vectors $\tilde{X}(t_i)$ and $\bar{X}(t_i)$ the value of the loss function $\mathcal{L}$ for each pair is computed at the same moment of time t_i;
4. The average error value is computed; this might be either the average or weighted average error value:
   (a) error $= \frac{1}{Q-1} \sum_{i=1}^{Q} \mathcal{L}\left(\tilde{X}(t_i), \bar{X}(t_i)\right)$,
   (b) error $= \frac{1}{Q-1} \sum_{i=1}^{Q} \alpha_i^* \mathcal{L}\left(\widetilde{X}(t_i), \bar{X}(t_i)\right)$,
5. The gradient of the error function is determined using the TensorFlow library's automated differentiation methods;
6. The gradient is used in the optimization of weights.

Additional improvements of the training results can be achieved by applying proper data normalization and regularization. These procedures are outlined below and applicable to both single-output models and multi-output models.

4.1 Regularization of Weights, Masking of Gradients

The usage of masks in the learning process is related to the necessity to keep matrices R^i physical in problems that depict a physical process, as well as the need to shorten the training time of a polynomial neural network. This functionality may be applied during training by zeroing and fixing tiny components of the weight matrix. Simultaneously, by omitting zero elements of the weight matrices from the training process, the number of configurable (trainable) parameters (weights) of the model is decreased.

To obtain a mask, we must first determine which of the coefficients is non-zero. This is only allowed if the matrices' values are received at the startup step. In this example, masks R_m^i of the same dimension as the matrices R^i are built so that there are *True* in place of non-zero elements of the original weights and *False* in place of zeros (or near to zero).

If the matrices Ri were not initialized when the model was built, the regularization L_0 of weights can be used to add to the loss function $\mathcal{L}$ the following value

$$\beta = \lambda \sum g\left(r_{jk}^{i}\right). \tag{4}$$

Here, *lambda* is the regularizing parameter that must be specified, $g(x) = \begin{cases} 1, x \neq 0 \\ 0, x = 0 \end{cases}$, r_{jk}^{i} are elements of matrix R^i. The norm L_0 penalizes the number of non-zero elements in the weight matrices, increasing their sparsity.

Because the function's combinatorial nature (4) and non-differentiability make optimization using traditional numerical algorithms impossible, various methods are proposed to weaken the discrete nature of the penalty [6,9,10] and apply optimization methods to smooth functions.

4.2 Normalization of Data and Initializing Weights

The weight matrices and data are adjusted to improve the learning outcomes of a polynomial neural network. The normalizing coefficient is computed using the norm L_infty for the vector of the of the initial data.

$$\|X\|_{\propto} = \max|X_i| \tag{5}$$

At the same time, for normalized data, a relation similar to (1) should be preserved

$$\tilde{\boldsymbol{X}}\left(t_{i+1}\right) = \text{bias} + \sum_{j=1}^{N} \widetilde{R}^{j} \tilde{\boldsymbol{X}}\left(t_i\right)^{[j]},$$

where ˜means normalization.

The algorithm for normalizing the weight matrices is as follows

1. Calculating the norm by the formula (5)
2. Multiplying all the elements of the matrix R^i by the coefficient $\alpha_i = \left(\|X\|_{\propto}\right)^{i-1}$.

5 Results of the Experiment

Let us consider an example system of ordinary differential equations Lotka–Volterra:

$$\begin{aligned} \frac{dx}{dt} &= (\alpha - \beta y)x, \\ \frac{dy}{dt} &= (-\gamma + \delta x)y \end{aligned} \tag{6}$$

with fixed parameters values $\gamma = \beta = 0.9$, $\alpha = 0.5$, $\delta = 1$. Let us illustrate the ability of PNN to describe this system for initial values not presented in the training data. According to the algorithm, presented in [8], we calculate the weight matrices W_k with low accuracy. We use the system (6) to generate

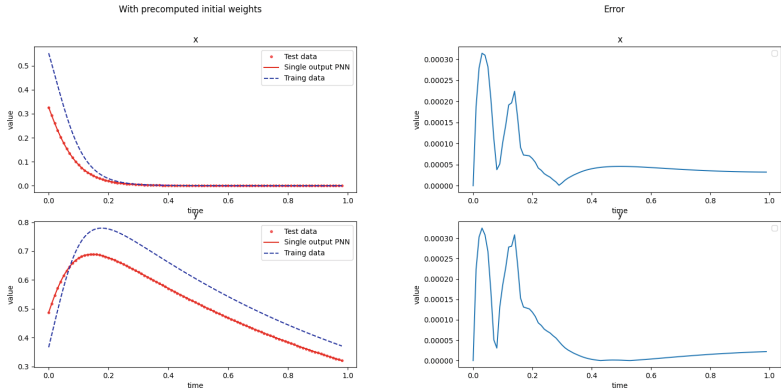

Fig. 5. Training results With precomputed initial weights

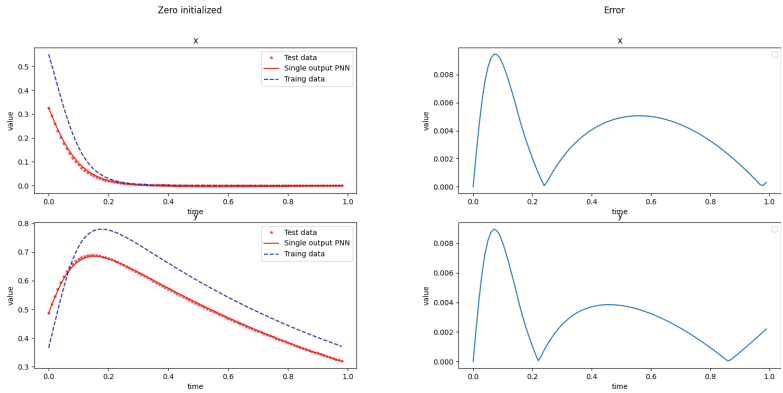

Fig. 6. Training results with zero initialization

the training and test data trajectories. We fit single output model which was initialized by matrices W_k and without initialization.

Figure 5 and 6 depict the results of system's trajectory prediction the integration of the Eq. (6) are presented after training the matrices in two ways compared with the test data. The absolute error at each moment of time is also computed. It can be seen that the maximum absolute error in the case of zero initialization does not exceed 5% and in the case of precomputed initial weights does not exceed 0.01%.

6 Conclusion

The paper contains the description of a learn module of tm-flow library for building and training polynomial neural networks to model dynamical processes. It also explains several learning strategies to increase speed and accuracy or

predictions, as well as approaches to keep the weight matrices physical, which allows you to boost the explainability of the outcomes. Such polynomial neural network can represent a general solution of ordinary differential equations, which permits canonical transformations to be preserved across large time periods. The outcomes of tests comparing training with and without initialization are given.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E.: TensorFlow: large-scale machine learning on heterogeneous systems (2015). https://www.tensorflow.org/
2. Al-Jumeily, D., Ghazali, R., Hussain, A.: Predicting physical time series using dynamic ridge polynomial neural networks. PLoS ONE **9**(8), e105766 (2014)
3. Chollet, F.: Deep Learning with Python. Manning Publications Co., New York (2018)
4. Dubey, A., Radenovic, F., Mahajan, D.: Scalable interpretability via polynomials, June 2022
5. Golovkina, A., Kozynchenko, V.: Neural network representation for ordinary differential equations. In: Dolinina, O., et al. (eds.) AIES 2022. SSDC, vol. 457, pp. 39–55. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-22938-1_3
6. Huang, C.T., Chen, J.C., Wu, J.L.: Learning sparse neural networks through mixture-distributed regularization. In: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), Seattle, WA, USA, pp. 2968–2977. IEEE, June 2020
7. Ivakhnenko, A.G.: Polynomial theory of complex systems. Trans. Syst. Man Cybern. **4**, 364–378 (1971)
8. Klimenko, I.S.: Implementation of the matrix mapping method for solving a system of differential equations. In: Control Processes and Stability, Saint Petersburg, Russia, vol. 9, pp. 53–57 (2022)
9. Liu, J., Xu, Z., Shi, R., Cheung, R.C.C., So, H.K.H.: Dynamic sparse training: find efficient sparse network from scratch with trainable masked layers, April 2020
10. Louizos, C., Welling, M., Kingma, D.P.: Learning sparse neural networks through L_0 regularization, June 2018
11. Oh, S.K., Pedrycz, W., Park, B.J.: Polynomial neural networks architecture: analysis and design. Comput. Electr. Eng. **29**(6), 703–725 (2003)
12. Shin, Y., Ghosh, J.: The pi-sigma network: an efficient higher-order neural network for pattern classification and function approximation. In: IJCNN-1991-Seattle International Joint Conference on Neural Networks, vol. 1, pp. 13–18 (1991). https://doi.org/10.1109/IJCNN.1991.155142