Extending Microservice Model Validity using Universal Differential Equations

Heimerson, Albin; Ruuskanen, Johan

Published in:
IFAC-PapersOnLine

DOI:
10.1016/j.ifacol.2023.10.1214

2023

Extending Microservice Model Validity using Universal Differential Equations

Albin Heimerson, Johan Ruuskanen

Abstract

When creating models of a system, there is always a trade-off between the ease of modelling a part and the increased value it brings to the model. Learning a model using machine learning, we might have less control over what dynamics to capture, but can also capture things we don't necessarily understand. Using universal differential equations we can combine the two, taking scientific models and embedding machine learning into them, with the goal of giving us the best of both worlds. In this paper, we extend an existing model with small neural networks to capture missing dynamics. The specific use-case involves a microservice fluid model, where the learned extension improves the range of parameters the combined model can reliably produce predictions over. This means the model can be reused in a wider parameter space before needing to be retrained. We also explore the possibility of imposing bias on the network based on features of the model, and how that affects performance.

Both authors are members of the ELLIIT Excellence Center at Lund University. We also thank Ericsson Research for letting us run experiments in their datacenter.

1 Introduction

The success of deep learning is often accredited to a few different things. Top among those are the generality of neural networks (NNs) as efficient function approximators, as well as large amounts of data for training and the resources to run large scale training. Though, in addition to this, many of the large breakthroughs in deep learning community comes from new ways of imposing inductive bias onto the model. One example being convolutional neural networks, with the idea that pixels close to each other were more likely to be related, or recurrent networks for time-series where the networks propagate information in one direction through time. Another bias could be imposed when looking for solutions to systems we expect to behave according to some differential equations (DEs). By letting the NN learn the dynamics instead of the solution, we can use well established DE solvers to simulate forward in time to find the solution. Taking this a step further, we can combine known dynamics with NNs to create a DE where parts are modelled using classical methods, and the missing parts are modeled by the NN. A differential equation that can model arbitrary dynamics is known as universal differential equations (UDEs).

This work specifically looks at models of cloud applications, and how we can improve them. There is a tendency in industry moving towards adoption of microservices as the framework for cloud applications, as shown in the survey by [1]. Microservices are a modern way of designing application in the cloud, where small independent services are each responsible for some objective within the application. They can be built, deployed and controlled individually, and thus give a large flexibility in the orchestration of the application. There is little centralized coordination, leading to easier control, but also leaving out a lot of possibilities for optimization. In [2] we looked at creating a control strategy by combining fluid models of a microservice application with automatic differentiation, optimizing the load balancing parameters by gradient stepping, and doing so with respect to the full context of the microservice application. We use gradients to update the parameters, but we only take a single bounded step before collecting new data, since we don't trust the model's prediction too much when the parameters change.

In this work, we look at *extending* existing ODE models with neural networks (NNs), to be able to make use of prior models while still improving upon them. This allows us to trust the model in a wider range of parameters, meaning more gradient steps can be taken before the model needs to be retrained.

1.1 Data-driven Discovery of Differential Equations

Using NNs to identify dynamics within an ODE has been done since the early 1990s, e.g. [3, 4, 5], as well as other function approximators such as Gaussian processes [6, 7, 8] or sparse regression methods [9, 10].

All of this can be unified under the term Universal Differential Equations (UDEs), where the dynamics of the DEs are modelled using universal function approximators of some form, allowing them to capture arbitrary dynamics.

$$\frac{d\boldsymbol{x}}{dt} = f(\boldsymbol{x}, t, \boldsymbol{\eta}) \tag{1}$$

Here we assume some function f parameterized by $\boldsymbol{\eta}$ defining the dynamics of the system, though we can easily say that $f(\boldsymbol{x}, t, \boldsymbol{\eta}) = F(\boldsymbol{x}) + NN_{\boldsymbol{\eta}}(\boldsymbol{x})$, to both introduce a good bias with the known dynamics in F, and capture whatever is missing with $NN_{\boldsymbol{\eta}}$.

Recent interest in combining ODEs and NNs has led to work such as [11, 12], producing tools for automatic differentiation of DE solvers, thus allowing efficient methods for solving these types of problems in continuous time.

Similar techniques have previously been used to model microservice applications, e.g. in [13] they used pure NN models to capture the discrete dynamics of a microservice mesh and predict future states to optimize resource allocations.

This is different to our work in that we attempt to keep the knowledge from an existing model, and simple add a small piece modelling the missing dynamics. In our work, we specifically want to keep the existing microservice model from [14], where the original model is good in most scenarios, though there are cases when running outside some of the system parameters where it was trained where it can have drastically reduced accuracy. This could be approached by training multiple versions of the original model, one model per parameter value, but as the system increase in size this can quickly become infeasible. By using a single base model extended by an NN, we can let the NN capture how the dynamics change from these parameters by training on data collected from a small amount of the parameter space. We evaluate the training efficiency and accuracy for the fluid model as well as the fluid model extended with an NN and compare them. We also explore how small inductive biases in the NN, based on our assumptions about the problem, affect the learning process and the final result.

2 Microservice Fluid Model

The fluid model for microservice applications used in this work is adopted from [14]. In it, each microservice can consist of multiple replicas, where each is modeled as a multiclass processor sharing queue. The delay between replicas is modeled as a multiclass delay queue. The service time for each class is allowed to follow a *phase-type* distribution. Furthermore, load balancing is assumed to follow a weighted random strategy, allowing the entire application to be expressed as a queuing network with probabilistic routing. The mean queue lengths of this queuing network can subsequently be approximated and quickly evaluated using the fluid model obtained via a refined mean-field approximation.

Let $\mathcal{Q}$ be the set of queues, i.e. all replicas in the microservices and their in-between delays. $\mathcal{C}_q$ is the set of classes for queue q, i.e. all classes of requests q can process. Then $\mathcal{S}_{q,c}$ is the set of phases from the phase-type service time in class c of queue q. Let k_q be the number of processors in queue q, $\boldsymbol{\lambda} \in \mathbb{R}_+^{|\mathcal{C}|}$ the Poisson arrival rate to each class and $\boldsymbol{P} \in \mathbb{R}^{|\mathcal{C}|\times|\mathcal{C}|}$ the class-to-class routing probabilities. Let $\boldsymbol{\Psi} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}$, $\boldsymbol{A} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{C}|}$ and $\boldsymbol{B} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{C}|}$ be the parameter matrices for the phase-type distributions for each class stacked into block diagonals. The model then approximates the average request population in each phase state $\boldsymbol{x}(t) \approx \mathbb{E}[\boldsymbol{X}(t)]$ as

$$\frac{d\boldsymbol{x}}{dt} = F(\boldsymbol{x}, \boldsymbol{\lambda}, p) = \boldsymbol{W}^T \theta(\boldsymbol{x}) + \boldsymbol{A}\boldsymbol{\lambda} \tag{2}$$

where $\boldsymbol{x}(0) = \boldsymbol{X}(0)$, $\boldsymbol{W} = \boldsymbol{\Psi} + \boldsymbol{B}\boldsymbol{P}\boldsymbol{A}^T$, and the function $\theta_i(\boldsymbol{x}) = x_i g_{Q(i)}(\boldsymbol{x})\ \forall i \in \mathcal{S}$ where

$$g_q(\boldsymbol{x}) = \frac{1}{\left(1 + \left(k_q^{-1} \sum \boldsymbol{x}_q\right)^{z_q}\right)^{1/z_q}} \qquad q \in \mathcal{Q}. \tag{3}$$

2.1 Extracting Metrics

Important performance metrics, such as population in classes and queues or response time percentiles, can be derived from the solution $\boldsymbol{x}(t)$ to (2).

The *mean requests* present at time t in the different classes and queues, can be approximated by summing over the corresponding phase states.

$$\begin{aligned} x_c(t \mid \boldsymbol{x}(t)) &= \sum_{i \in \mathcal{S}_c} x_i(t) \\ x_q(t \mid \boldsymbol{x}(t)) &= \sum_{i \in \mathcal{S}_q} x_i(t) \end{aligned} \tag{4}$$

Let $\boldsymbol{x}_C \in \mathbb{R}_+^{|\mathcal{C}|}$ and $\boldsymbol{x}_Q \in \mathbb{R}_+^{|\mathcal{Q}|}$ be the corresponding vectors of all mean requests for classes and queues respectively.

The *response time percentile* can be approximated by setting up another ODE based on $\boldsymbol{x}^*$, the steady state solution of (2). Let $\boldsymbol{\pi}(t) \in \mathbb{R}_+^{|\mathcal{S}|}$ be the probability vector of finding a request in the corresponding phase state after time t, and $\beta \in \mathbb{R}_+^{|\mathcal{C}|}$ the probability vector of the request entering the corresponding class at time $t = 0$. The probability of the request remaining in the network can then be approximated with the ODE

$$\frac{d\boldsymbol{\pi}}{dt} = \boldsymbol{W}^T D^g \boldsymbol{\pi}(t), \qquad \boldsymbol{\pi}(0) = \boldsymbol{A}\beta \tag{5}$$

where $D^g \in \mathbb{R}_+^{|\mathcal{S}|\times|\mathcal{S}|}$ is a diagonal matrix with elements $D^g_{ii} = g_{Q(i)}(\boldsymbol{x}^*)$. The approximation of the percentile φ_α can then be found by evaluating (5) until $\sum \boldsymbol{\pi}(\varphi_\alpha) = 1-\alpha$ holds, or by doing, e.g., a bisection search over its closed-form solution. We denote this solution as $\varphi_\alpha(\boldsymbol{x}^*)$.

3 Method Description

3.1 Identifying base fluid model

We extract the base fluid model using the latest batch of data according to the method described in [14]. The extracted model is then used to simulate the load in the system using (2), as well as estimating the expected 95th percentile response time using (5). To predict outside the current parameter settings, the $\boldsymbol{P}$ matrix can be updated with desired load balancing values, and $\boldsymbol{\lambda}$ can be changed to reflect expected incoming load. This updated model can now be simulated to get expected loads and response time percentiles for a different parameter setting. Though this model is accurate around the parameter setting where it was fitted, it will have an increased tendency for prediction errors as we get further away from the parameters used for the data collection.

To decrease these prediction errors, we add two parts to the model. An NN modelling the missing dynamics in the class population from the fluid model, and an NN modelling the error of the response time prediction.

3.2 Modelling missing dynamics

We set up our extended model using as a universal ordinary differential equation, where a small neural network is added to the fluid model to capture the missing dynamics.

$$\frac{d\boldsymbol{x}}{dt} = F(\boldsymbol{x}, \boldsymbol{\lambda}, p) + NN_{\boldsymbol{\eta}}(\boldsymbol{x}, \boldsymbol{\lambda}, p) \tag{6}$$

The neural network is parameterized by $\boldsymbol{\eta}$ in addition to depending on the same parameters as the fluid model, i.e., the phase-states $\boldsymbol{x}$, arrival rate $\boldsymbol{\lambda}$ and load balancing probability p. It is trained by simulating the ODE until convergence, and comparing the steady-state class population $\boldsymbol{x}^*_C$ to the recorded average class population $\hat{\boldsymbol{x}}_C$ via the mean squared error. We do this over some historical data $\mathcal{D}$

$$L_C(\boldsymbol{\eta}) = \sum_{p,\boldsymbol{\lambda},\hat{\boldsymbol{x}}_c \in \mathcal{D}} (\boldsymbol{x}^*_C(\boldsymbol{\lambda}, p) - \hat{\boldsymbol{x}}_C)^2 \tag{7}$$

where $\boldsymbol{x}^*_C(\boldsymbol{\lambda}, p)$ is the steady state solution collected into different classes according to (4), and $\boldsymbol{\eta}$ are the parameters of the NN. We estimate the steady state solution by simulating for some time t_f that is large enough that the system normally will have reached steady state, for this system $t_f = 2$ s was found to be sufficient. Further, we use $\boldsymbol{\lambda}$ from recorded data when training, since this will allow a better fit and will likely create a more accurate model.

How the networks are designed can have a large effect on learning efficiency and accuracy, and imposing bias on them can improve learning speed as well as accuracy if done right. The basic NN contain only dense layers with linear activation in the output layer, and will be referred to as `nn` or `fluid nn` depending on if we use the fluid model as base.

A slightly more informed model includes a final layer that makes sure the NN does not create or remove any flow, just change the internal flows between the phases-states. This can be implemented for the NN by simply subtracting the mean of the last layer from itself, thus creating an output that sums to zero. We will refer to this method as `fluid nnfc`.

$$fc(\boldsymbol{z}) = \boldsymbol{z} - \frac{1}{|\mathcal{S}|}\sum_{i=1}^{|\mathcal{S}|} z_i. \tag{8}$$

A third approach constrains the flow based on the fluid model, i.e. looking at the rows in $\boldsymbol{W}$ where each row representing internal flows sum to zero, we call this `fluid nnmc`. By having the output of the final layer of the NN be weights that each set a flow between two phase states in the pattern defined by $\boldsymbol{W}$, the flow is forced to follow the pattern of the fluid model. As long as the fluid model

is general enough, this should simplify learning while not reducing the accuracy of the model.

Using the models to estimate future behavior for different load balancing parameters, we don't know what the future $\boldsymbol{\lambda}$ will be since it is a stochastic variable. We choose to estimate it by the most recently recorded value, as that is likely a good approximation of the future value.

3.3 Modelling response time prediction error

The improvements to the fluid model will already make the estimate of the response time percentile (5) better, since we now have more accurate predictions for the class populations. But the main problem of the assumption that each request receive the mean processor share as mentioned in [15] still remains. In most cases, this will be less true the larger the utilization due to the increased variability in queue lengths. The error of (5) can thus be expected to correlate with the utilization. Due to this correlation with the solution of the ODE, it becomes simple to improve the estimate. We do so by adding a small NN to estimate the error of the solution, which is trained in the step after the UDE model is trained.

Having the predicted $\boldsymbol{x}^*$, the phase state population in steady state, we can then estimate the response time percentiles from (5), and calculate the errors compared to the recorded values. Now we fit a small regularized network using supervised learning to map from the load on each replica, to the error of the estimation in (5).

$$\begin{aligned} \varphi_{err} &= \hat{\varphi}_\alpha - \varphi_\alpha(\boldsymbol{x}^*(\boldsymbol{\lambda}, p)) \\ L_{\varphi_\alpha}(\boldsymbol{\mu}) &= \sum_{p,\boldsymbol{\lambda},\hat{\varphi}_\alpha \in \mathcal{D}} (\varphi_{err} - NN_{\boldsymbol{\mu}}(\boldsymbol{x}_Q^*(\boldsymbol{\lambda}, p)))^2 + \alpha \sum_{\mu \in \boldsymbol{\mu}} \mu^2 \end{aligned}$$

where $\hat{\varphi}_\alpha$ is the response time percentile estimated directly from the data and $\boldsymbol{\mu}$ are the parameters for this neural network.

3.4 Control

One use-case for these improvements is to optimize a cost function for the distributed application operations with respect to the load balancing parameters. In [2], the cost function is conditional on whether the limit for φ_α is violated, selecting control parameters to either minimize the response time or to minimize the cost to run the distributed load.

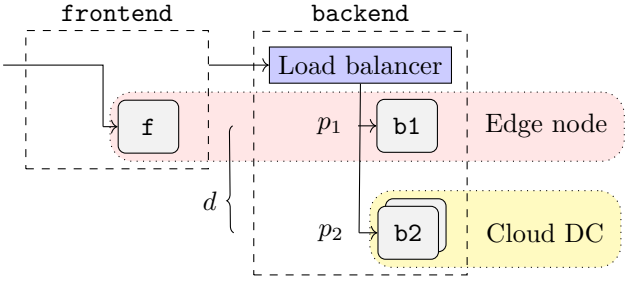

Figure 1: A simple distributed application with two microservices. The `frontend` only exists close to the user on the edge nodes, and the `backend` has replicas on both the edge nodes and the cloud. The backend has a weighted random load balancer with routing probabilities $p_1 = p$ and $p_2 = 1 - p$. There is a delay d between the edge and the cloud.

$$L(p) = \begin{cases} C_\varphi \varphi_\alpha \left(\boldsymbol{x}^*(\boldsymbol{\lambda}, p)\right) & \text{if } \varphi_\alpha > \varphi_{\text{lim}} \\ \boldsymbol{C}^T \boldsymbol{x}_Q^* \left(\boldsymbol{\lambda}, p\right) & \text{otherwise} \end{cases} \tag{9}$$

4 Experiment description and Results

4.1 Application

We consider a distributed microservice application as shown in Fig. 1 which is subject to external requests with exponential inter-arrival times, i.e. Poisson arrivals, with the rate parameter $\lambda = 14$. It is a simple face detection service consisting of two connected microservices, a frontend and a backend, where the backend is made up of two replicas distributed over different sites. Each instance are running with 4 cores, i.e. $k_q = 4 \forall q \in \mathcal{Q}$. A load balancer using a weighted random scheme controls how jobs are distributed over the backend, deciding which replica is to service any specific request. There is a real computational cost which will be affected by where a load is placed and what type of load it is, as well as a cost for breaking the response time constraints. For this experiment, the coefficients for the computational $\boldsymbol{C}$ is set to be a zero vector except for the two backends where it was set to $\boldsymbol{C}_b = [3, 1]$, and the cost coefficient for the response time constraint is set to $C_\varphi = 8$. We use $\alpha = 0.95$ to look for the 95'th percentile response time as well as the limit $\varphi_{lim} = 0.55$. There is also a communication delay

d between the sites which affect soft constraints such as response time percentiles. All connections between the two clusters are given a Pareto distributed additive delay with a 25ms mean, 5ms *jitter* (approximately standard deviation) and 25% correlation between samples.

4.2 Data and Training

The application used to evaluate the improvements was run on a federated cluster using the tools introduced in [16].

In this work, the cluster was only used for recording real data for the experiments, and all training and verification was done offline using the recorded data. This is not a restriction of the method in general, but was more convenient for experimentation. In [2] we show that the corresponding methods transfer well from offline evaluation to the live application.

The data is collected as 5-minute recordings, containing all arrival and departure logs for each request. From these logs the class populations can be extracted, and this data is collected for all $p \in [0, 0.05, \ldots, 1]$. For these experiments, we use $p = 0.6$ as the current load-balancing parameter, i.e. the data the fluid model is trained on. The NN can additionally use data collected at $p \in [0.1, 0.3, \ldots, 0.9]$, which together with data from $p = 0.6$ forms the training data $\mathcal{D}$. The remaining data is only used to visualize how well the model captures the parameters in general. The historical data was selected in a sparse grid over the interval to have a chance of finding a good model for the missing dynamics.

The training is done using the SciML (Scientific Maching Learning) ecosystem in Julia which has DE solvers that support AD. This means that we can define a function that, based on some initial parameters, solves the DE, evaluates a loss based on the solution and generates the gradient of the loss with respect to the parameters. The NN optimization is done using the ADAM algorithm, a standard first order method commonly used in deep learning. All the code is publicly available on Github[1].

4.3 Evaluating model designs

We compare three different NN extensions to the fluid model, as presented in Section 3.2, to see how imposing bias on the net affects the learning speed as well as the accuracy of the learned model. For comparison, we also show the performance of the fluid model by itself, as well as a purely NN based model, though using the same amount of states in the UDE as for the fluid model.

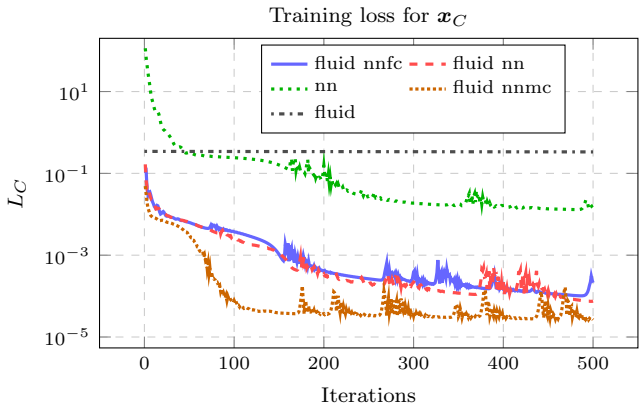

Figure 2: Training loss for the different models defined in Section 3.2, and the loss of the fluid model as reference.

For the fluid model, we give each class in the queuing network 3 phase states, for a total size of $|\mathcal{S}| = 24$. For the NN only and extended models, the dense layers of the neural networks are all the same, three hidden layers with 20 units each and exponential linear units for activation.

Fig. 2 show how the loss L_C converge during training for the different methods. The presented data is the average over 5 trials using different seeds for the training.

The models using only NNs had some problems learning the dynamics, and would sometimes either produce an unstable system, crashing the ODE solver, or not finding the relationship between the class dynamics and the load-balancing parameter p, as can be seen in Fig. 4. If trained for even longer, they sometimes found a decent model, though they had a larger tendency to overfit to data instead of capturing the dynamics.

Comparing the models that extend the fluid model, we see that they are quite similar. All of them start around the same performance as the fluid model, and improve from there. The extended model using an NN that has its output constrained by the dynamics of the model, `fluid nnmc`, does however converge quite a bit faster than the other ones. Seeing this, we will evaluate the performance in more detail using only `fluid nnmc` out of the extended models, though we keep both `nn` and `fluid` to compare against.

[1] `https://github.com/albheim/IFAC2023_code`

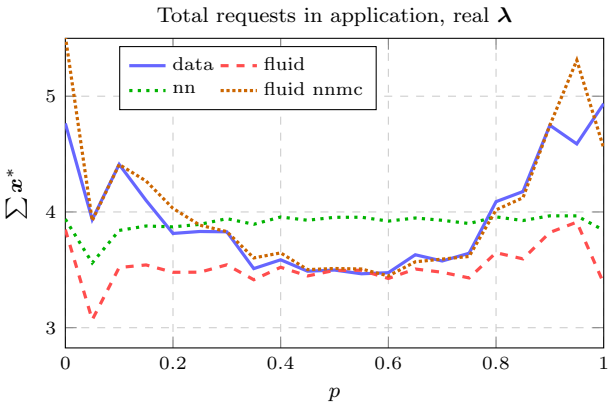

Figure 3: Here we use the recorded $\boldsymbol{\lambda}$ for each evaluated p, making for a better comparison of how well we fit the model, instead of testing the predictive capability.

4.4 Evaluating performance metric predictions

To start with, we look at how well the extended model captures the actual data by supplying the real $\boldsymbol{\lambda}$ in the whole range, see Fig. 3. The fluid model captures the real data well around $p = 0.6$ where it was trained, though as we move further away by updating p and $\boldsymbol{\lambda}$ in the model, we see the prediction accuracy decline. The model with only NN does not do too well, and seems to not have captured much of the dynamics. The extended model is a lot more accurate and actually matches the data quite well, though we still see some artifacts, e.g. at $p = 0.95$ where the data decrease but all models increase. This can also be seen in the fluid model, and is likely an artifact from building upon it.

In Fig. 4 we see the predicted request population, the predicted response time percentile and the predicted cost for the fluid model as well as the flow constrained NN model for different number of training iterations. The requests are based on a Poisson process, and the load balancer is random, so the real data show some stochastic behavior that the model does not predict. This is mainly from $\boldsymbol{\lambda}$ not being known for the values we predict, and instead estimated based on the most recent data, so here we use the $\boldsymbol{\lambda}$ that was estimated for $p = 0.6$. We can clearly see this difference when comparing the predictions for the queue-length in Fig. 3 and 4. To estimate the cost function well, the response time percentiles play a big role, especially when close to its limit. As seen in Fig. 4, neither the fluid model nor the model with only an NN

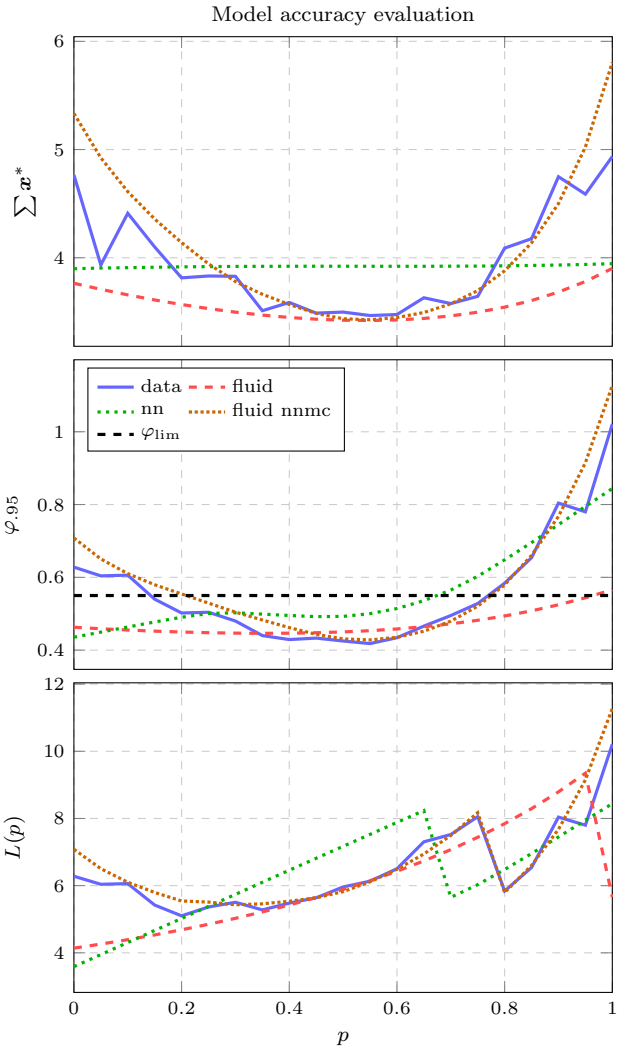

Figure 4: Here we see the recorded data as well as predictions from a few different models. From the top we have the total number of requests, the 95th percentile response time and the cost function.

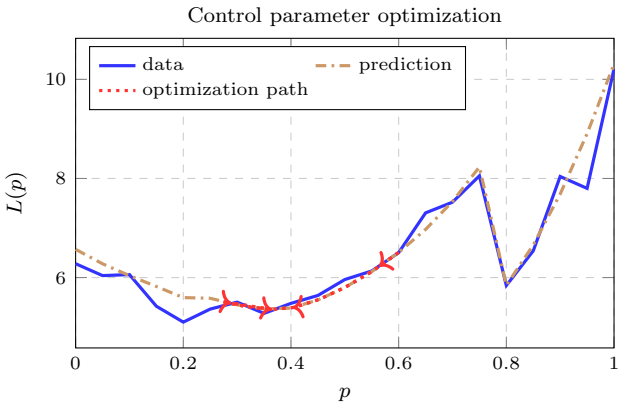

Figure 5: Gradient descent on p based on the loss of the trained `fluid nnmc` model. Starting at $p = 0.6$ which is the recent data used for the fluid model, and ending up around $p = 0.3$.

predict this well, resulting in a cost where the optimum is at $p = 0$. The extended fluid model identifies a minimum around 0.3, close to where the data also show it.

4.5 Control

To visualize this in a control scenario, Fig. 5 shows how the control parameter is optimized using the model prediction. Using the model from Fig. 4, we run gradient descent starting at $p = 0.6$ on the predicted values from the model. Exactly where the real minimum is will vary since the load is stochastic, but we can see that the found p ends up reasonably close to an optimal setting for the real data.

5 Conclusion

The type of models used in [2] can be extended to have more accuracy outside the current parameter settings by adding a small NN to the model in order to capture the missing dynamics. The experiments in this work show how the efficiency of training can increase by providing basic dynamics in the model, and how these models are more likely to ultimately capture correct dynamics. In addition to using an existing model as base, imposing structure from the model onto the NN itself is also explored. Both this and using the base model are different ways of adding bias to the NN, and for a good bias we are likely to see improving training efficiency and model accuracy. We analyze different combinations of these biases, and in our experiments both of them improves learning speed and accuracy for our moderately sized networks.

These models are combined with the techniques for control presented in [2], to take more gradient steps with the same model and achieve faster convergence. In this way we can optimize the control parameters for the combined model, ending up close to the actual optimal value, with a single trained model. For the fluid model this is not feasible, as it would end up at $p = 0$ given the loss in Fig. 4. To achieve the same results, it would instead require smaller steps where new data was collected and trained on in-between, resulting in slower convergence to good control parameters.

The method itself is more general than the specific use-case presented here, and could be applied to a wide variety of applications by combining some basic knowledge about the system with UDEs to model the missing parts. Using gradient based methods to find optimal parameters for control is only one option for control, though a very convenient one if the framework for AD exists.

5.0.1 Future work

We would like to run this for a larger system, where it would be infeasible to train and store a model for each parameter setting. This would also require running it against a live system since we would not collect a fine enough grid of data for a larger system. To make this feasible, we need to improve the runtime performance of the algorithm, and one way to achieve this could be to translate the implementation to use reverse-mode differentiation.

References

[1] Steve Swoyer and Mike Loukides. Microservices adoption in 2020, Jul 2020.

[2] Albin Heimerson, Johan Ruuskanen, and Johan Eker. Automatic differentiation over fluid models for holistic load balancing. *Presented at ACSOS 2022. Email albin.heimerson@control.lth.se for access.*, 2022.

[3] Ramiro Rico-Martinez, K Krischer, IG Kevrekidis, MC Kube, and JL Hudson. Discrete-vs. continuous-time nonlinear signal processing of cu electrodissolution data. *Chemical Engineering Communications*, 118(1):25–48, 1992.

[4] Raul González-García, Ramiro Rico-Martìnez, and Ioannis G Kevrekidis. Identification of distributed parameter systems: A neural net based approach. *Computers & chemical engineering*, 22:S965–S968, 1998.

[5] Zichao Long, Yiping Lu, Xianzhong Ma, and Bin Dong. PDE-net: Learning PDEs from data. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 3208–3216. PMLR, 10–15 Jul 2018.

[6] Michael Schober, David K Duvenaud, and Philipp Hennig. Probabilistic ode solvers with runge-kutta means. *Advances in neural information processing systems*, 27, 2014.

[7] Maziar Raissi and George Em Karniadakis. Hidden physics models: Machine learning of nonlinear partial differential equations. *Journal of Computational Physics*, 357:125–141, 2018.

[8] Markus Heinonen, Cagatay Yildiz, Henrik Mannerström, Jukka Intosalmi, and Harri Lähdesmäki. Learning unknown ODE models with Gaussian processes. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1959–1968. PMLR, 10–15 Jul 2018.

[9] Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the national academy of sciences*, 113(15):3932–3937, 2016.

[10] Samuel H Rudy, Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Data-driven discovery of partial differential equations. *Science advances*, 3(4):e1602614, 2017.

[11] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. *CoRR*, abs/1806.07366, 2018.

[12] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, and Ali Jasim Ramadhan. Universal differential equations for scientific machine learning. *CoRR*, abs/2001.04385, 2020.

[13] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. MIRAS: Model-based Reinforcement Learning for Microservice Resource Allocation over Scientific Workflows. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 122–132, July 2019.

[14] Johan Ruuskanen and Anton Cervin. Distributed online extraction of a fluid model for microservice applications using local tracing data. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 179–190. IEEE, 2022.

[15] Johan Ruuskanen, Tommi Berner, Karl-Erik Årzén, and Anton Cervin. Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing. *Performance Evaluation*, 151:102231, 2021.

[16] Johan Ruuskanen, Haorui Peng, Alfred Åkesson, Lars Larsson, and Maria Kihl. Fedapp: a research sandbox for application orchestration in federated clouds using openstack, 2021.