Evaluation of Deep Learning architectures for System Identification. Applications for a household heating system

Cristian Vicas
Computer Science Department, *Technical University of Cluj-Napoca, Romania*
cristian.vicas at cs.utcluj.ro

***Abstract*—Model Predictive Controllers are widely used in the industry to control a dynamic system. They require a good model for the controlled process. In this paper I try to perform system identification using promising current neural network architectures. I frame the problem as a regression setup but allow for future possible commands to be integrated in the response. I test six architectures on three datasets. I adapt successful neural network architectures from Computer Vision and Natural Language Processing to predict the output of a system for some future time steps, given current knowledge plus some future commands. Results are promising but the models are not ready to be integrated in a real time, resource constrained environment.**

***Index Terms*—system identification, deep learning, model predictive control**

I. Introduction

There are several ways in which a fairly linear process can be controlled. Most popular ones are PID (Proportional, Integral, Derivative) and for systems with significant delay, Smith controllers or better, Model Predictive Controllers (MPC). Both Smith and MPC controllers rely heavily on a good model of the controlled system [1]. This model must output predictions for time $t_1, t_2, \cdots$ given the current state at t_0 and a set of future commands, $c_1, c_2, \cdots$.

Classical system identification consist in building a mathematical model (more or less complex and accurate) for the real system and then deduce its parameters by sending a specially constructed set of commands to it.

Rather than stimulating the system with various impulses I try to mimic the real system from existing commands and responses using neural networks (NN). The NNs have another advantage: they can forecast certain periodical behaviors of the inputs. So if the system is perturbed by external factors, under certain assumptions NNs can predict their evolution. These predictions are intrinsic and are incorporated into the output of the system model.

Once one have such a system model, (that is able to predict future evolutions given an explicitly set of commands and probably intrinsic perturbation predictions) one can use various optimizations or search strategies to find the best set of commands with respect to some loss. Note that integrating the model in such algorithms is not in focus here.

There are few drawbacks in using neural networks in control. The most important one is the "black box" nature of the networks [2]. While the knowledge of how a NN reasons about the inputs is interesting, the more important knowledge that comes with an explainable system is the failure mode. How and when a NN will fail is still an active area of research.

In this paper the following assumptions are made, about the modeled system:

- The system has an existing closed loop control.
- The system has a delayed response to the commands.
- Some relevant inputs, outputs and internal signals are monitored with enough temporal resolution and enough data is recorded.
- There are other factors that influence the system, some of them are not monitored.
- There are irrelevant or marginally relevant inputs in the data.
- Analytical modeling is hard (eg. too complex or missing inputs).
- At t_0, past sensory data plus commands are available $(t_{-n} \dots t_{-1})$.
- The space for possible commands is rather small (eg. $c \in \{0, 1\}$) and a certain temporal correlation is imposed on these commands so the space for all the possible future commands (on a finite window) is small.

Microcontrollers, actuators and sensors have become more and more available to the point where one can "drag and drop" software + hardware components to create a rather complex system. However, the needed know-how on how to model and control the created system follows an opposite direction. Also sensors are noisy, actuators have transient failures, mechanical systems (especially cheap ones) have lower tolerance specifications. All these factors complicate the system to a point where it is not practical to analytically model it.

Machine Learning field follows the same trend of becoming more and more accesible. Several devices have dedicated matrix multiplication units (the core operation for some very powerful techniques) or chip accelerators that allows one to translate a rather fancy model on a low power embedded device. Edge Computing and TinyML are also some buzzwords that are starting to emerge.

In this paper I show a method to circumvent the analytical modeling problem by using Machine Learning (ML). Second, (because there is no free lunch [3]) I show some mainstream architectures and approaches from ML that might perform well for physical system modeling. One of the main contributions of this paper is to show how to adapt NLP/CV architectures to a time series problem where both past and future information is available. I do not intend to replace a PID or MPC controller with ML but improve such a system.

This paper brings several contributions to the field:

- It tries to commodify modelling with the help of Machine Learning.
- It frames a System Identification problem as a ML problem.
- It adapts several well known ML architectures to work with specific data characteristics needed for MPC-like controllers.
- It tests the architectures on two simulated datasets and one dataset from a real system.
- Code and some data is available online [4]. Keen reader can apply these models to their data, fast.

II. Related work

There are toolboxes for the ML that implement proven state-of-the-art methods emerging every few months. Also, a lot of techniques and tricks are surfacing. Unfortunately the whole "Deep Learning" (DL) subfield is in alchemy mode (trial and error are still the base of the progress). Except for some elements (eg loss functions) there is no valid theory on why and how the DL actually works. The whole domain relies heavily on tricks and hacks to boost the learning performance [5], [6]. To be clear, DL works very well and in some fields there are no methods in sight that are challenging their performance (Computer Vision and Natural Language Processing, to name two fields where DL methods dominate all the benchmarks).

While the popular belief is that NNs are "general function approximators" the reality is that without strong priors the DL will overfit and produce useless approximation of the data. It is well known in the DL field that these priors known as inductive biases are baked into the DL architectures sometimes explicitly or sometimes are discovered post-hoc [7]. One could train a general enough network but the data requirements and computational power are intractable.

There are several papers that rely on a ML/DL model to do system identification problem. In [8] the author uses a form of LSTM embedded into a convex optimizer to do system identification on some simulated systems. In [2] the authors take a rather rare form of neural netowrk (General dynamic neural networks) and apply it to several simulated instances. It is possible to directly derive the PID control parameters using ML [9] or [10]. In [11] the authors apply a different ML/optimization branch, the Evolutionary programming paradigm.

It is noted that whatever model is present inside the controller it must be able to handle delayed signals (for current decision, the relevant signal was recorded several steps ago) more than one signal, and/or to be able to predict how a control signal will affect the system without actually triggering that signal.

In present paper I study several popular DL architectures that might have the enumerated inductive biases. In Natural Language Processing (NLP) past information is vital for the current prediction, especially for translation (so called sequence to sequence problem). An interesting mechanism was proposed in [12], called *the attention* where at current time t_0 the network learns what kind of inputs from $t_p < t_0$ are relevant to compute current output. By coupling several attention mechanisms together (eg. maybe one point in the history is not enough to explain the current predicted value) authors in [13] developed Transformer architectures. In [14] they managed to tackle the realm of symbolic mathematics.

Another inductive bias is the locality and translation invariance (consecutive measurements have roughly the same meaning and the relative offset with respect to some arbitrary origin is irrelevant). These biases are baked into convolutional architectures (CNN). CNNs can handle delays because CNNs are usually layered in stacks.

Recurrent neural networks were available for some time now but they were hard to train due to numerical instabilities. These issues are largely solved and now. One can use off the shelf LSTM or GRU layers without worrying much about these problems [15]. From the sequence to sequence field I also noted the encoder-decoder approach where one network "translates" the inputs to some lower dimensionality space and then another network will take this encoding and generate desired outputs.

Unfortunately one can't directly use all these models for our problem statement. NLP problems have discrete inputs/outputs (words, letters) and this assumption is strong in learning strategies like *teacher forcing*. The CNN must be applied only timewise; there is no *a priori* ordering on the features so the CNN layer must ignore one spatial dimension and focus only on temporal dimension. In present paper I try to handle all these aspects.

III. Problem description

A. Formal definition

Given:

(a) a set of vectors $x_1 = \{a_0, \ldots, a_k, b_0, \ldots, b_l, c_0, \ldots, c_m\}$ each of the a, b, c vectors of real numbers, containing time series for sensor information, commands and outputs from a system;
(b) a sub set of commands $x_2 = \{ c_i \cdots c_j\}$ that are in the future of the x_1 vectors;
(c) a sub set of sensor information, $y = \{b_i, \cdots b_j\}$ that are temporally aligned with x_2 vectors.

We create a function f that will predict the subset of values y that, when the real system is presented with the x_2 commands will record those values: $f(x_1, x_2) = y$.

We want to create a model that takes: different sensor values (including some output parameters that are monitored and that we care about), prior commands given to the system (to control those output parameters) and some potential future commands that we intend to give to the system. Based on this input the model will output how the values that we care for will vary in the future.

This paradigm can be used as a system model inside a MPC control loop.

B. System

My system is a household with a central heater, radiative and convective heating elements and outside walls. Fig. 1 shows the heating circuit. *H* is the heating unit that has a pump *P1* with On/Off switch. The circuit directly feeds a radiator *R1* and through *P2* feeds a high inertia radiative element *R2* (floor heating). Temperature sensors (green dots) are mounted on various locations. Sensor te_5 monitors the ambient temperature and te_6 outside temperature.

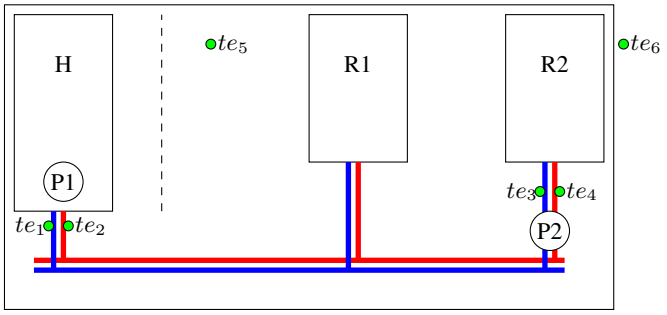

Fig. 1. Studied heating system. Details in the text.

There is a data collection unit and more temperature sensors and actuators monitored than those shown in Fig. 1. However, those depicted in Fig. 1 are essential for our problem.

C. Data

In this paper I evaluated the architectures on three datasets: one real dataset and two simulated datasets.

Real data. There are 14 features in the data that represent various sensors and actuator statuses sampled at 1 minute interval. Besides the important sensors shown in Fig. 1 (te_1 to te_6) there are temperature sensors on tap water, attic and other heating radiators in the chain that do not directly affect the air temperature in the household (vectors x_1 in section III-A). The controlled (or predicted) variable (vectors y in section III-A) for this system is sensor te_5 (inside air temperature).

I collect information about the commands issued to the heater and recirculating pumps (*P1* and *P2* from Fig. 1). The most important control point is *P2* so this will be the future command sub set (vector x_2 in section III-A)

The data is recorded over approx. 3 year period (*Y0-September* to *Y2-May*). For training and evaluation I selected several data regions listed in Table I. *Winter* means from September previous year to April current year. So *Winter Y1* means *September Y0* to *April Y1*.

Because of various hardware conditions there are randomly missing values and sometimes (eg sensor failure) continuous periods without a signal. I manually selected regions that have low missing variable count.

Simulated data. There are two sets of simulated data and both have the same "structure" as the real dataset. First simulated dataset, *sim0* is very simple: the features are random and the output is a linear combination between one input feature and the information about the future that is available at present time. The last feature in the input shows which of the other features contribute to the output. For: a time length n, $f+1$ features, $k < f$ indicator variable then the f'th feature will have the portion $\lfloor k * (n/f) \rfloor - \lfloor (k+1) * (n/f) \rfloor$ biased with a positive value.

To predict some valid output for *sim0* the model must learn: that there is valid information in the past, that its location is "described" by one of the input features and that the future information is also relevant for the output.

The second set, *sim1* follows the structure and the meaning of the real data. Three features are relevant: outside temperature, inside temperature and issued commands. The inside temperature is computed as a sum of delayed and convoluted outside temperature and heating commands. Let $a(t)$ be the inside temperature at time t, $e(t)$ outside temperature, and $c(t) \in \{0, 1\}$ heating commands. A very crude heating equation is:

$$\begin{aligned} a(t) = a(t-1) + \\ \alpha_e \left((k_e \circ e)[t - T_e] - a(t-1) \right) + \\ \alpha_c (H_c - a(t-1))(k_c \circ c)[t - T_c] \end{aligned} \tag{1}$$

The $\circ$ is the convolution operator, [] indexes into the array, k_e and k_c are two Gaussian kernels, α_e and α_c are scaling values and T_e, T_c are some delays. Note that the heating agent is always at the same temperature H_c and gradually heats up the air with a delay T_c.

A successful model must learn how to combine temporal information from the past with available future data and to discard random features. Moving to ML framework, the training instance is a tuple (x_1, x_2) with target y. Past information is available as features in x_1. Vector x_2 has the same temporal length as y and consists in possible future commands. Again, how to generate those commands is not in focus here. For the **real** dataset there is a closed loop control system that generates these commands. For **sim1** these commands are random.

D. Data conditioning

Data windowing is applied to the *sim1* and *real* datasets. The continuous 1 minute signals are resampled with a larger time base and then cutted in fixed size windows. First part of the window with length S, with $F = 14$ features, will build a training sample. From the second part of length T, the $te5$ column is the target for the learner. I extract the P2 signal from the second part and fed it as an input. The model f has to minimize the root mean squared error between the predicted values and target values:

$$rmse = \sqrt{\sum \frac{(f(x_1, x_2) - y)^2}{N}} \quad (2)$$

where N is the number of samples, $x_1 \in \mathbb{R}^{F \times S}$, $x_2 \in \mathbb{R}^T$ and $y \in \mathbb{R}^T$. For Transformers architecture, x_2 has a special shape $x_2 \in \mathbb{R}^{F2 \times T}$ where $F2 = 2$. $x_2[0, \cdot]$ is the regular P2 command feature but the $x_2[1, \cdot]$ is the target variable y shifted with one position to the right: $x_2[1, k+1] = y[k]; k < T$ and $x_2[1, 0] = -1$.

Data normalization. Most networks have a batch normalization layer directly at the input [16]. The only special handling is for the target y where I subtracted the average of the last known $te5$ values from the target features y. As a result, the network has an easier job to predict values close to zero without having to learn some offset. Of course, at inference time, this offset is added back to the predictions and all the statistics regarding the performance are computed with respect to the original values.

TABLE I
TRAINING AND EVALUATION DATA FROM *real* DATASET. NOTE THAT THE VALIDATION DATA IS IN THE FUTURE OF THE TRAINING DATA.

Name	Period	Count	Observations
train	Winter Y1	6K	Main train dataset
train-large	Autumn Y1 - Autumn Y2	18K	Extended dataset including spring and summer of Y2.
train-mild	Winter Y1	3K	Days with outside temp. <-10°C were removed
validation	Winter Y2	8K	Main validation set
val-cold	Winter Y2	0.4K	Only days with outside temp. < -10°C are included.
val-quiet	Winter Y2	0.2K	Few days without human presence.
val-night	Winter Y2	1K	Included only night intervals between 23PM and 06AM

E. Architectures

In the following, *B* denotes the mini-batch size, *S* is the number of available past time steps and *T* is the length of the future sequences. *H* denotes the size of some "hidden" layers inside the architecture. Usually *F* is the number of input features for each timestamp.

First architecture (named *Linear*) is a simple linear combination between concatenated inputs. There is no activation function but the inputs are normalized before entering the linear layer. This will act as a baseline. If another model performs worse than this model, it is probably overfitting. Also, if the error level of *Linear* model is good enough, the problem is simple.

The Computer Vision workhorse and probably the brute force approach here is shown in Fig. 2. The input is convoluted time-wise with a kernel of size 7 followed by zero or more convolutions with kernel size 3. The number of channels in the intermediate layers is kept constant and is a hyperparameter. Everything is flattened and appended to the secondary input (x_2) then fed to a stack of Dropout and Linear+ReLu layers. The last layer has no activation function and will predict in one shot the output. I call it *CnnFcn* throughout the paper.

The third architecture *Enc-Dec* is a simple encoder-decoder based on stacked LSTM layers. There is no attention implemented here and the decoding inputs are x_2 concatenated with the hidden states of the encoding stack. A hyperparameter is the number of LSTM layers and another is the dropout value between layers.

The fourth architecture called *Att* shown in Fig. 5 is an encoder (Fig. 3) followed by a decoder (Fig. 4) with *attention* mechanism. *Attention* uses the known information at step $k-1$ to "focus" the network on inputs (or encodings) relevant to step k. I used the variant proposed by [12] with some adaptation to current problem. I appended the known facts about the future (x_2) in the inputs for step k of the decoder. The first hidden state of the decoder is the encoder hidden sate. The predictions are done iteratively, one k value at a time. The $k-1$ predicted output is then fed as known input into the decoder. Note that there is no *teacher forcing* mechanism implemented here. The output passes through a *tanh* activation before it is fed to the last linear layer. After the attention is applied to the encoder states and this "focused" attention is concatenated with the current input, the data passes through a *Batch Norm* layer. This turned out to be vital for architectures with more than one recurrent layer. *Post-hoc*, I explain this by noting that the input is poorly represented (only two values) and the dynamic range is limited (esp for *P2* commands). Vanilla [12] without the specified changes does not perform any learning.

I also implemented two more architectures based on Transformer block [13]. First transformer variant uses the same approach as the sequence-to-sequence in *Att*. This Transformer block accepts during training both the known inputs and masked output. While there is a speed advantage at learning, the prediction must be done sequentially as for any sequence-to-sequence problem with *attention* architecture. I adapted it to fit the knowledge about the future samples and to cope with different feature lengths. The input is filtered with one 1D convolution stack (Batch Norm, 1D Conv, ReLu) with kernel size of 1 to expand the number of channels to the number of Transformer's hidden states. One Conv 1D operation is performed for the Transformer output to "shrink" the number of channels to 1. At prediction, the "embedded" output is iteratively fed back to the transformer in the corresponding position. The initial value for $k = 0$ of the input is a vector with -1 values. Because I use a full sequence-to-sequence approach I call it *TransF*. In *TransF*, *teacher forcing* is used at training. By using x_2 as targets and using the first "iteration" outputs directly (dropping the *teacher forcing*), I get a simpler version called *TransS*.

IV. RESULTS

I allowed each network to train for a large number of epochs, reducing the learning rate or stopping the learning altogether if the performance on the validation set did not improve. To control this, I used only *validation* set. The hyperparameters for each architecture were randomly sampled

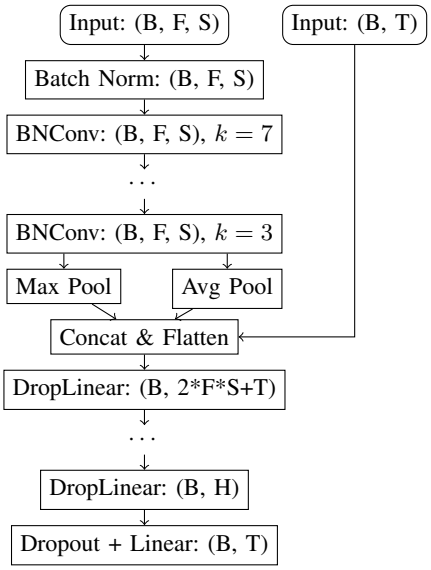

Fig. 2. *CnnFcn* architecture. BNConv layer is a stack of a 1D convolutional layer with k filters, a Batch normalization followed by ReLu nonlinearity. DropLinear is a stack of a Dropout layer, a Linear layer and a ReLu activation.

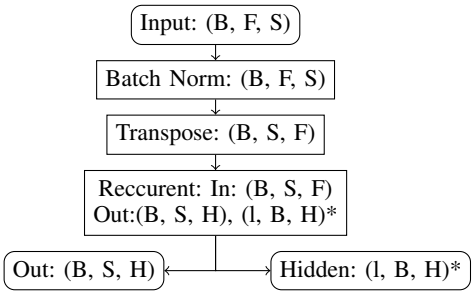

Fig. 3. Attention, encoding layer. Small *l* is the number of layers in recurrent layer. (*) The hidden output shape varies if I use LSTM, GRU or other type of recurrent architecture.

several times. To train I used *Adam* optimizer with learning rate annealing [17], [18]. After all the desired iterations were performed, each model was evaluated again on all four validation sets listed in Table I.

Simulated data. Dataset *sim0* was used to check the implementation of the models. I validated the fact that both the past and the available future information is picked up by the model. I plotted the predictions (after few training epochs) for most important architectures in Fig. 6. The *TransF* and *Att* architectures gave a rather flat response showing that there might be issues with their learning process.

I trained all the models using *sim1* dataset for few epochs and plotted a prediction for each model in Fig. 7. *Linear* model is fairly noisy with a lot of variability even if it roughly follows the data. The *Enc-Dec* and *Att* architecture fit the data rather well. For the rest of the architectures, the flat response could indicate either a bad fit for the data or that it needs more epochs to learn a useful signal. Unfortunately the results on the real dataset inclined toward the former.

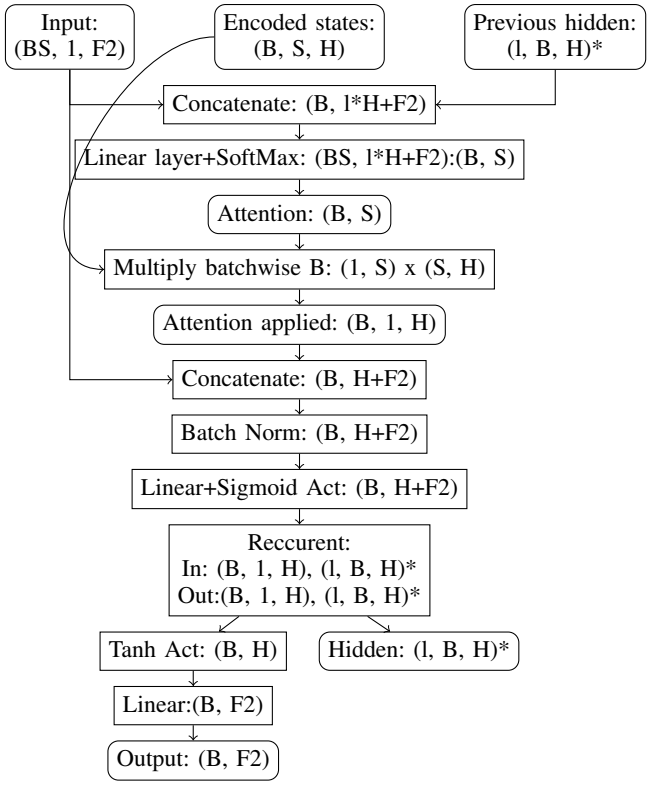

Fig. 4. Attention, decoding layer. The input is served one timestamp at a time. The second dimension from the input has size 1. The previous *hidden* input must be compatible with the input of the recurrent layer. To simplify things, both the encoder and decoder have the same recurrent architecture and the same number of layers.

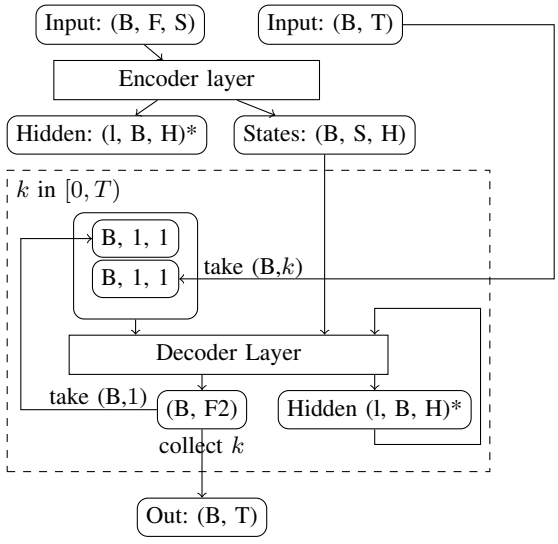

Fig. 5. Attention model.

I did not perform any further investigation on the simulated datasets.

Real data. To show the results, I will use *boxplots*. Each box encloses the 25-75 percentile. Whiskers are at 10-90 percentile and the dots are outliers. The horizontal line inside the box is the median value of the plotted data.

Fig. 8 shows the overall performance of the architectures evaluated on *validation* set. *Att* has good results and surprisingly the *CnnFcn* has consistent performance and lower loss values too. I visually inspect the quality of the data by plotting some predictions from each network. Fig. 9 shows the ground truth with some actual commands executed by the system. I plotted the predictions from each network for that sample.

The network was chosen as the best performer on the architecture class. To remove the clutter, I show in Fig. 10 only the architectures that managed to have models with low RMSE: *CnnFcn*, *Enc-Dec*, *Att* and *TransS*. To my surprise the full Transformers model did not perform well. Table II shows some training hyperparameters, training time and the size of the networks (number of parameters) for the best models in each category.

In the following I will focus only on the architectures that gave us best results, *CnnFcn*, *Enc-Dec* and *Att*.

I used several validation datasets, each one with different specifications. For each of the selected architectures and for each validation set listed in Table I I show in Fig. 11 the RMSE boxplots. I note that *CnnFcn* is constantly better than *Enc-Dec* while there is no clear difference between *CnnFcn* and *Att*.

The next question is what impact the data volumes have on the training? Also, I am curious on how the architectures extrapolate on unseen data ranges. To evaluate this extrapolation capability I have a training set that is lacking the cold winter temperatures (*train-mild*) and a validation set that includes only the cold period from the next year (*val-cold*). Fig. 12 shows this for *CnnFcn* architecture and Fig. 13 for *Att* architecture. The results are fairly similar between the architectures and the extreme values in the input have no impact on the prediction performance.

TABLE II
BEST MODEL HYPERPARAMETERS USED IN PREDICTION.

Architecture	Hyperparams	Size	Train time (seconds/epoch)
Linear	None	87K	3s
CnnFcn	CNN layers=1 FCN layers=3 CNN filters=60	20M	4s
Enc-Dec	Layers=2 Hidden size=256	1.5M	11s
Att	Layers=1 Hidden size=256	19K	89s
TransS	Hidden size=32 FCN size=64	26K	6s
TransF	Hidden size=128 FCN size=256	402K	70s

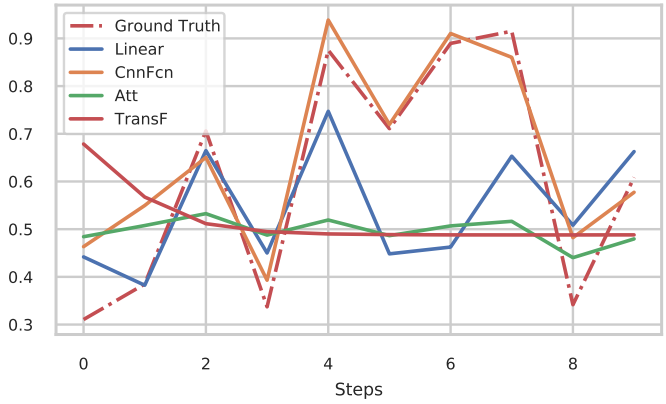

Fig. 6. Predictions on simulated data *sim0*. Each model was allowed to train for approx. 10-15 epochs. The ground truth is just a linear combination of some random input features. On *Oy* axis is some abstract value without physical meaning.

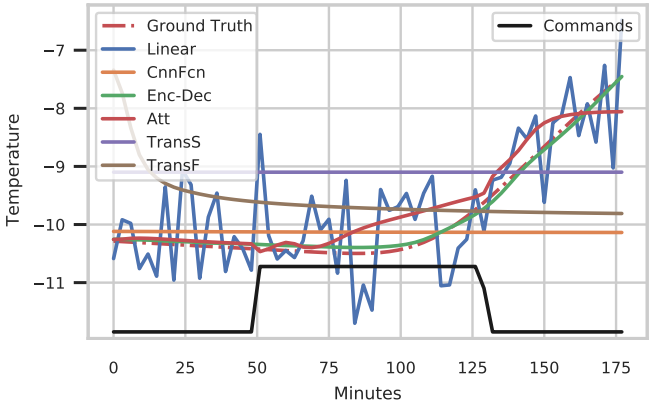

Fig. 7. Predictions on simulated data *sim1*.

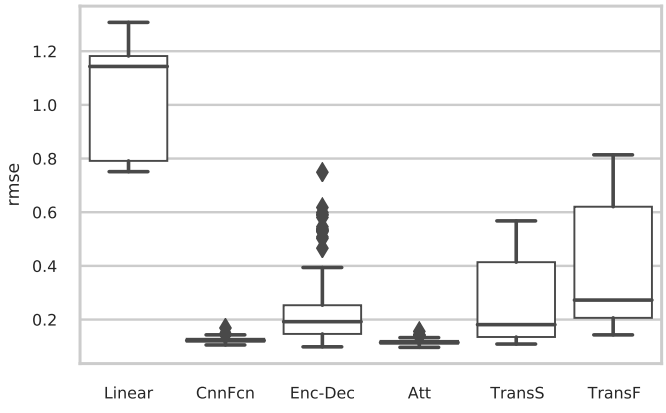

Fig. 8. Overall performance on *validation* dataset for all the architectures.

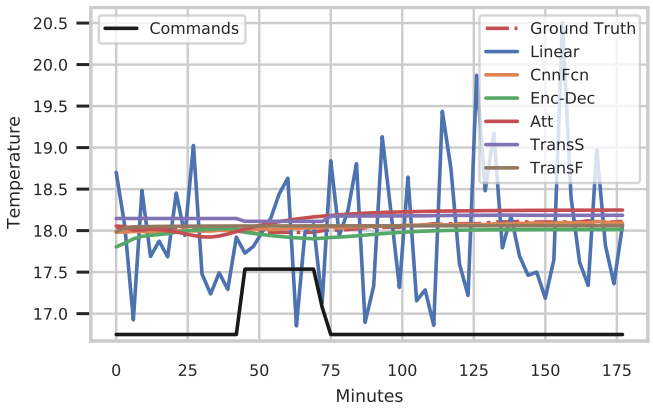

Fig. 9. Predictions on real data. The sample is from *valid-quiet* dataset. Note that the *Linear* architecture has a high variability and *TransF* has a constant response.

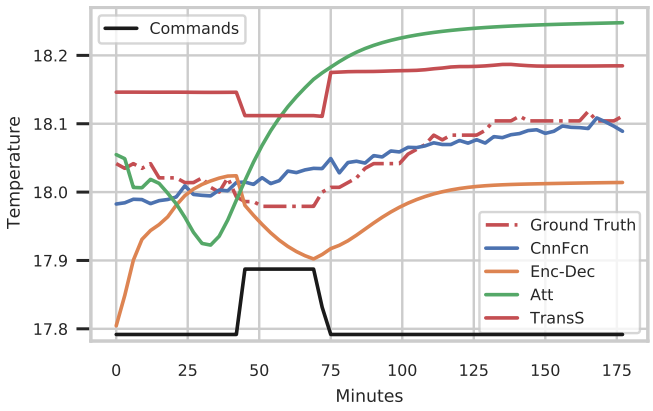

Fig. 10. Predictions on real data, same sample as in Figure 9 but without high variability results (*Linear* and *TransF*). The sample is from *valid-quiet* dataset.

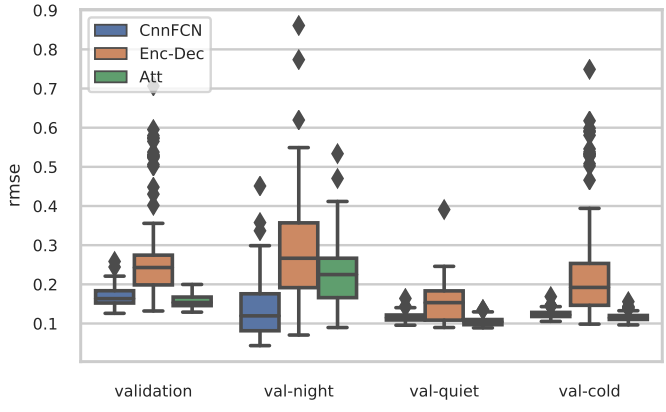

Fig. 11. Overall performance on validation datasets for selected architectures. RMSE values larger than 1 were removed because of some outliers in the results.

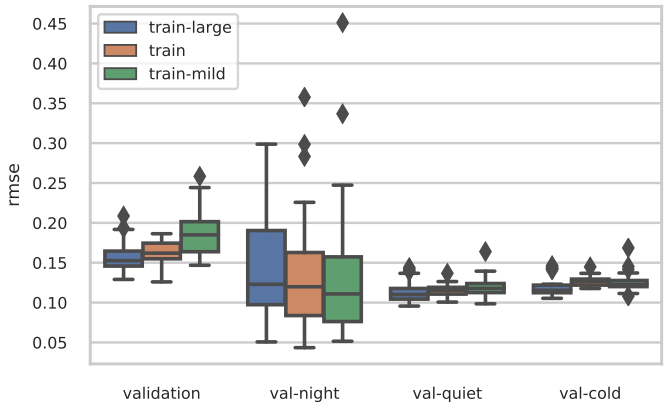

Fig. 12. Performance on validation datasets for *CnnFcn* architecture with respect to the training set size. RMSE values larger than 0.6 were removed because of some outliers in the results.

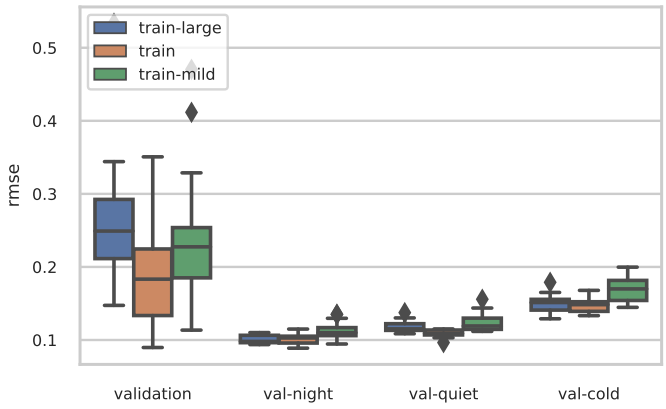

Fig. 13. Performance on validation datasets for *Att* architecture with respect to the training set size. RMSE values larger than 0.6 were removed because of some outliers in the results.

V. Conclusions

This paper acts as a baseline on what mainstream architectures can deliver for this particular problem.

It was surprising that a "general purpose" network performed similar to sequence oriented architectures. However, it is possible that the inductive biases that are implemented by recurrent neural networks are not fitted for our problem. The *softmax* layer that computes the *attention*, forces only one past timestamp to be relevant for current prediction step. Counterintuitive, changing this behavior by putting a *sigmoid* activation, rendered the network useless. One might try to experiment adding a temperature in *softmax*. Some authors reported this as an alternative to "fixating" the attention to only one sample in the past.

The LSTM methods and simple Transformers have apparently correlated outputs (at least on few studied samples). It is not clear why this happens. Even *CnnFcn* architecture could use some more in depth analysis and ablation studies. In Image Processing, stacking a Convolution, Batch Norm and

then ReLu layers is a standard practice. Is this true here?

Despite successful adaptation of *Att*, there is an empirical observation in the DL field: all the alterations and variants of an architecture have limited abilities to improve on the performance. Better results will be obtained by choosing the right architecture with the right inductive biases.

The need for larger training data (I have 26.000 instances) could be handled with Self Supervised Learning [19], [20] and simulated data. We saw that the best performing model had 20 milion parameters. In the future I will investigate model distillation [21], or mixed precision models.

There are some concerns regarding the outliers present in some evaluation datasets. This might be mitigated with different weight initialization strategies. The training times vary greatly between architectures. The *CnnFcn* trains faster than attention based strategies but with the same performance. In inference time, especially in embedded devices that lack massively parallel multiplication cores, a more supple network is desirable (eg *Att* with 19K parameters and with close performance to 20M *CnnFcn*).

It is well established in the automation community that some features must enter the model with a delay. One could explicitly include this in the network, creating some sort of feature engineering and then allow the network to "choose" what features to use and with what delay. This is similar to how Inception like architectures work [22].

In this form, I can't really say that the problem I approached is solved. This paper is just a step on a longer road.

The most important conclusion is that practitioners could benefit from ML in control problems. I will focus next on finding the right (or better) inductive biases for this particular control problem and investigate more systems.

REFERENCES

[1] M. R. Stojic, M. S. Matijevic, and L. S. Draganovic, "A robust Smith predictor modified by internal models for integrating process with dead time," *IEEE Trans. Autom. Control.*, vol. 46, pp. 1293–1298, 2001.

[2] J. Günther, E. Reichensdörfer, P. M. Pilarski, and K. Diepold, "General Dynamic Neural Networks for explainable PID parameter tuning in control engineering: An extensive comparison," *CoRR*, vol. abs/1905.13268, 2019. [Online]. Available: http://arxiv.org/abs/1905.13268

[3] D. H. Wolpert, "The Lack of A Priori Distinctions Between Learning Algorithms," *Neural Computation*, vol. 8, no. 7, pp. 1341–1390, 1996.

[4] [Online]. Available: https://github.com/cristi-zz/auto_iccp2020

[5] T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie, and M. Li, "Bag of Tricks for Image Classification with Convolutional Neural Networks," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 558–567.

[6] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256.

[7] P. Battaglia, J. B. C. Hamrick, V. Bapst, A. Sanchez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. Allen, C. Nash, V. J. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks," *arXiv*, 2018. [Online]. Available: https://arxiv.org/pdf/1806.01261.pdf

[8] Yu Wang, "A new concept using LSTM Neural Networks for dynamic system identification," in *2017 American Control Conference (ACC)*, 2017.

[9] A. Zulu, "Towards explicit PID control tuning using machine learning," in *2017 IEEE AFRICON*, 2017.

[10] S. Genc, "Parametric system identification using deep convolutional neural networks," in *2017 International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 2112–2119.

[11] A. Nyberg, "Optimizing PID parameters with machine learning," *ArXiv*, vol. abs/1709.09227, 2017.

[12] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2014.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[14] F. Charton, A. Hayat, and G. Lample, "Deep differential system stability – learning advanced computations from examples," *arXiv*, 2020.

[15] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[16] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 448–456.

[17] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *3rd International Conference on Learning Representations*, vol. abs/1412.6980, 2015.

[18] L. N. Smith, "Cyclical Learning Rates for Training Neural Networks," in *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2017, pp. 464–472.

[19] J. Schmidhuber, *Making the world differentiable: on using self supervised fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments*, ser. Forschungsberichte Künstliche Intelligenz. Inst. für Informatik, 1990. [Online]. Available: https://books.google.ro/books?id=9c2sHAAACAAJ

[20] I. Misra and L. van der Maaten, "Self-Supervised Learning of Pretext-Invariant Representations," *ArXiv*, vol. abs/1912.01991, 2019.

[21] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," in *NIPS Deep Learning and Representation Learning Workshop*, 2015. [Online]. Available: http://arxiv.org/abs/1503.02531

[22] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.