Preprints, 10th IFAC International Symposium on
Advanced Control of Chemical Processes
Shenyang, Liaoning, China, July 25-27, 2018

A Deep Learning Architecture for Predictive Control

Steven Spielberg Pon Kumar * **Aditya Tulsyan** *
Bhushan Gopaluni * **Philip Loewen** *

* *University of British Columbia, Vancouver, Canada (e-mail: spiel@mail.ubc.ca; tulsyan@ualberta.ca; bhushan.gopaluni@ubc.ca; loew@math.ubc.ca).*

Abstract: Model predictive control (MPC) is a popular control strategy that computes control actions by solving an optimization problem in real-time. Uncertainty and nonlinearity of a process, and the non-convexity of the resulting optimization problem can make online implementation of MPC nontrivial. Consequently, MPC is most often used in processes where the time constants are large and/or high-performance computing support is available. We propose a deep neural network (DNN) controller architecture to reduce the computational cost of implementing an MPC. This is done by training a DNN controller on simulated input-output data from a well-designed MPC. The online implementation of a DNN controller does not require solving an optimization problem. Once the DNN is trained, the MPC is fully replaced with the DNN controller. The benefits of this approach are illustrated through a simulated example.

Keywords: artificial intelligence, model predictive control, deep neural networks, optimization

1. INTRODUCTION

Developing a reliable control strategy for complex industrial processes in presence of process uncertainties is a daunting task. This problem is further exacerbated in presence of non-linear and non-stationary dynamics, known or unknown process disturbances, and multivariate process interactions. Model predictive control (MPC) is an advanced control strategy that addresses these challenges. For a detailed survey of the MPC and its applications, see (Qin and Badgwell, 2003; Yu-Geng et al., 2013).

The central idea behind the standard MPC is straightforward – it computes a control action by minimizing a cost function, e.g., tracking error, over the space of feasible actions. In a standard MPC formulation, the optimization is solved repeatedly, taking into account the latest observations, the system dynamics (as represented by a process model), and other input-output constraints. See Mayne (2014) for details.

Even though MPC has been successfully applied in industrial processes worldwide, its implementation for complex processes is still lagging. This is primarily due to the challenges in effectively solving the optimization problem in real-time with nonlinear dynamic models and other non-convex constraints. Solving the optimization problem becomes even more challenging in the presence of hidden nonlinear Markov models, as in nonlinear MPC (NMPC) (Findeisen et al., 2007) or with economic objectives, as in economic MPC (EMPC) (Ellis et al., 2014). Several global methods exist to solve complex non-convex constrained optimization problems with dynamic models (Biegler, 2000, 2007); however, these methods are not practical for online implementation due to their high computational cost.

The lack of an effective approach to solve constrained optimization problems in real-time has made MPC algorithms suitable for processes either with large time constants or for environments with high-performance computing (HPC) capabilities. For a number of fast dynamic systems in areas without the HPC capabilities, such as manufacturing, electrical, and aerospace, available optimization algorithms do not yet meet the requirements of real-time computing. Thus, despite recent theoretical advances, MPC is not widely used in large-scale applications in these fields (Yu-Geng et al., 2013).

To make the MPC implementaiton practical for complex systems, researchers have proposed many techniques to facilitate real time computing. For example Wan and Kothare (2003) use *off-line design and online synthesis* strategy to design a constrained robust MPC controller from Kothare et al. (1996). This strategy moves a significant portion of computation offline. While it has a solid theoretical basis, the algorithm needs to off-line solve an NP-Hard multi-parametric programming problem. Wallace et al. (2016) propose a two-tiered optimization approach to speed up the optimization step, where the first tier computes the best response of certain desired form like fast first order response with minimum overshoot ratio and the second tier computes the control action to achieve this fast response. However, they suffer from the curse of dimensionality. Various function approximation techniques are also developed for use with MPC. Kittisupakorn et al. (2009) use Neural Networks (NN) to approximate the MPC prediction step and optimization cost function. NN are also used to approximate the nonlinear system dynamics, as in Piche et al. (2000), and then MPC is applied on the neural network model. However, these approaches involve estimation of hidden states at each instant. In ad-

dition, model free methods such as reinforcement learning techniques Pon Kumar et al. (2017) were also developed to speed up the optimization step. Our approach differs from the fact that our newly proposed deep learning architecture takes past control actions and current system output to compute the control action at each step. Also in our approach we have directly mapped from process output, target output to control actions. Once the deep learning model is trained, it does not require performing optimization step to return control action. Moreover, our approach does not involve estimation of hidden states that characterize system dynamics.

In this paper, we propose a novel approach to reduce the computational cost of implementing an MPC. The central idea is as follows. First, for a given process and process constraints, we design *the best-in-class* MPC architecture that achieves optimal controller performance by way of selecting the *best* process model, the *best* state estimator and the *best* optimization method. Clearly, such an deal MPC would have an optimal performance, but limited practical use. Next, under simulations or controlled implementation, pairs of control actions and system outputs are generated using the MPC. The generated data include information about the policy implemented by the MPC by solving the complex constrained optimization to global optimality. Next, using the generated data, a neural-network (NN) model is trained to emulate the control policy implemented by the ideal MPC. Once a NN is trained, the MPC is discarded, and the NN is deployed for real-time control.

The proposed method offers several advantages. First, unlike a traditional MPC, the NN controller does not solve any expensive constrained optimization problems, state estimation algorithms or model simulations in real-time. This makes it extremely fast, and suitable for real-time control. Second, since the NN controller emulates the control policy implemented by an ideal MPC, its performance is also expected to closely follow that of an ideal MPC. Our simulations demonstrate the efficacy of the proposed deep NN controller in controlling a simulated process.

To ensure that the NN generalizes to complex processes with large number of continuous states, we develop a deep NN architecture for the controller using *Long Short-Term Memory Supported NN* (LSTMSNN) models. The LSTMSNN is a weighted combination of the standard LSTM with NN. The LSTM part of the architecture uses past control actions to capture the temporal relationships between the control actions, while the NN part uses current system output and target output to predict the control actions. This ensures that the current control action depends on the past control actions, the current system output and the target output. To accelerate the online implementation of the deep NN controller, it is implemented in a Graphical Processing Unit (GPU) by parallelizing the computations.

Finally, we highlight that in this paper, we are not proposing any novel designs for MPC. Instead, the user designs and selects the MPC they wish to use to control their processes. The proposed method simply takes the user-designed MPC, and makes its implementation computationally less demanding through a NN controller.

2. NEURAL NETWORKS

The process modeling problem can be formulated as a regression problem. The objective is to find a mapping function $f\colon \mathcal{U} \to \mathcal{Y}$ that best describes the process dynamics, where $\mathcal{U} \subseteq \mathbb{R}^{n_u}$ and $\mathcal{Y} \subseteq \mathbb{R}^{n_y}$ are the input and output spaces, respectively.

The underlying principle behind the use of NNs to model complex process dynamics is the adaptation of a natural network of neurons, where each single neuron predicts an output, $y_t \in \mathcal{Y}$, by forming a linear combination of *truths* constructed from the inputs, $u_t \in \mathcal{U}$, and applying a nonlinear transformation to produce an output prediction, $\widehat{y}_t \in \mathbb{R}^{n_y}$. In a single-layer neural network, the relation between y_t and u_t can be expressed as

$$y_t = f_y(W_{uy}u_t), \tag{1}$$

where $W_{uy} \in \mathbb{R}^{n_y \times n_u}$ is a weight matrix from u_t to y_t, and $f_y(\cdot)$ is an element-wise nonlinear activation function. Common activation functions include the sigmoid, hyperbolic tangent, and rectified linear unit:

$$f_y(x) \equiv \sigma(x) = \frac{1}{1+e^{-x}}, \tag{2}$$

$$f_y(x) \equiv \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}, \tag{3}$$

$$f_y(x) \equiv \mathrm{ReLU}(x) = \max(0, x). \tag{4}$$

Networks are formed by composition. For example, a 3-layer NN model with sigmoid activation has the functional form

$$y_t = W_{3,uy}\sigma(W_{2,uy}(\sigma(W_{1,uy}u_t))), \tag{5}$$

where $W_{1,uy}, W_{2,uy}$ and $W_{3,uy}$ are matrices of network weights. Combining a large sum of these simple neurons across the input space, and forwarding them through multiple layers in a network, provides a versatile class of functions from which one with the desired properties is to be selected. The selection process, called "learning", consists of choosing the network weight matrices to minimize a suitably defined loss function. Typically one minimizes the difference between the measured/desired input-output data and the NN predictions, using a gradient descent algorithm. Repeated application of the chain rule for differentiation leads to an efficient method for gradient calculation known as backpropagation.

2.1 Recurrent Neural Networks

Recurrent neural networks (RNN) extend conventional Feed Forward Neural Networks (FFNNs) by adding time-dependence. RNNs have been widely studied and used for various machine-learning tasks involving temporal (or spatial) correlations between the inputs and outputs. For example, the input and output sequences of a controlled dynamic system are time-series with temporal correlations, satisfying the Markov property. An RNN model captures the behavior of such systems using a hidden state, $h_t \in \mathcal{H} \subset \mathbb{R}^{n_h}$, whose dynamics are described as follows

$$h_t = f_h(h_{t-1}, x_t), \qquad y_t = f_y(h_t), \tag{6}$$

where $f_h : \mathcal{H} \times \mathcal{X} \to \mathcal{H}$ is a transition function (possibly nonlinear) describing the dynamics of $h_t \in \mathcal{H}$ and $f_y : \mathcal{H} \to \mathcal{Y}$ describes the functional mapping from the hidden state to the output variable. The hidden state, $h_t \in \mathcal{H}$, can

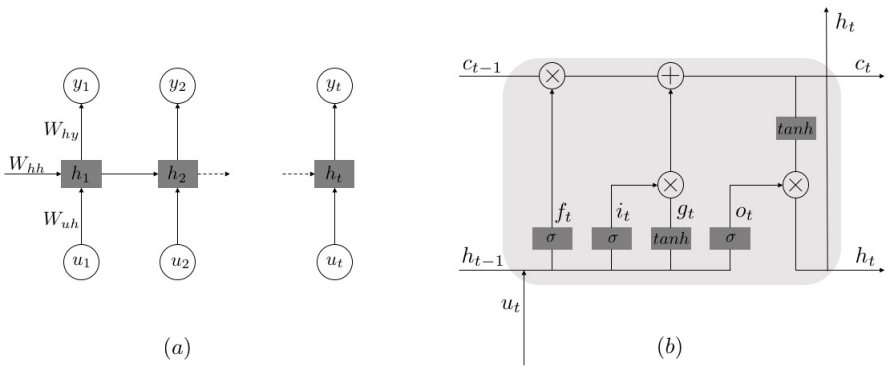

Fig. 1. The architecture of an (a) RNN and (b) LSTM. The figure is adapted from (Jithesh et al., 2017).

be interpreted as a running summary of $u_t \in \mathcal{U}$ until time $t \in \mathbb{N}$; the recurrence formula in (6) updates this summary based on its previous value, $h_{t-1} \in \mathcal{H}$. It is common to assume $h_0 = 0$.

The transition function f_h is typically defined as the composition of an element-wise nonlinearity with an affine transformation of both u_t and h_{t-1}, such that

$$h_t = f_h(W_{uh}u_t + W_{hh}h_{t-1}), \tag{7}$$

where $W_{uh} \in \mathbb{R}^{n_h \times n_u}$ and $W_{hh} \in \mathbb{R}^{n_h \times n_h}$ are the input-to-hidden weight matrix and hidden-to-hidden weight matrix, respectively. Similarly, f_y is typically defined as a composition of an affine transformation of h_t, such that

$$y_t = f_y(W_{hy}h_t), \tag{8}$$

where $W_{hy} \in \mathbb{R}^{n_y \times n_h}$ is the hidden-to-output weight matrix. It is common to define f_h as a sigmoid or hyperbolic tangent function, and f_y as the identity function, so

$$h_t = \tanh(W_{uh}u_t + W_{hh}h_{t-1}), \qquad y_t = W_{hy}h_t. \tag{9}$$

In (9), $W \equiv \{W_{uh}, W_{hh}, W_{hy}\}$ is a set of network parameters that can be estimated using backpropagation method through time (BPTT [21], [22]), whereby parameters are updated after a complete sequence of forward and backward passes are completed. RNNs have memory ability, but long-term memory is limited. It makes the gradient explode or vanish while training with BPTT algorithm, which is addressed in the approach of (Bengio et al., 1994).

2.2 Long Short-Term Memory

The challenges of training an RNN to capture long-term input-output dependencies are well-known to the research community. Several attempts have been made to address this fundamental problem, including modifying f_h to encourage some hidden units to adaptively maintain long-term memory and creating paths in RNN, such that gradients can flow over many time-steps.

The LSTM architecture was first proposed by Hochreiter and Schmidhuber (1997). Since then, it has been revised and refined by many other researchers. An illustration of an LSTM architecture is given in Figure 1(b). An LSTM unit consists of a *memory* cell, c_t, an *input* gate, i_t, a *forget* gate, f_t, and an *output* gate, o_t. The memory cell carries the memory content of an LSTM unit, while the three gates control the amount of changes to and exposure of the memory content. In Figure 1(b), u_t is the input to the memory cell layer and σ is the element-wise logistic sigmoid function. Mathematically, for n_p LSTM units, the forget, input and output gates are described as follows

$$f_t = \sigma(W_{uf}u_t + W_{hf}h_{t-1}), \tag{10a}$$
$$i_t = \sigma(W_{ui}u_t + W_{hi}h_{t-1}), \tag{10b}$$
$$o_t = \sigma(W_{uo}u_t + W_{ho}h_{t-1}), \tag{10c}$$

where $W_{uf} \in \mathbb{R}^{n_p \times n_u}$, $W_{ui} \in \mathbb{R}^{n_p \times n_u}$, and $W_{uo} \in \mathbb{R}^{n_p \times n_u}$ are the LSTM weights from the input, u_t to f_t, i_t and o_t, respectively. Similarly, $W_{hf} \in \mathbb{R}^{n_p \times n_h}$, $W_{hi} \in \mathbb{R}^{n_p \times n_h}$ and $W_{ho} \in \mathbb{R}^{n_p \times n_h}$ are the weights from the hidden state, h_{t-1} (at the previous time-step) to f_t, i_t and o_t, respectively. The gates in (10a)–(10c) control the information flow through the LSTM. The forget gate, f_t, determines how much of the hidden state, h_{t-1} is allowed to pass through; the input gate, i_t determines the past values of the hidden state, h_{t-1}, that will be updated; and the output gate, o_t, determines the how much of h_{t-1} will be made available to the next layer. Finally, the candidate value, g_t, and the memory cell, c_t, are updated via

$$g_t = \tanh(W_{ug}u_t + W_{hg}h_{t-1}), \tag{11a}$$
$$c_t = f_t c_{t-1} + i_t g_t, \tag{11b}$$

where $W_{ug} \in \mathbb{R}^{n_p \times n_u}$, and $W_{hg} \in \mathbb{R}^{n_p \times n_h}$ are the weights for u_t and h_t to g_t, respectively. Once c_t is updated in (11b), the hidden state h_t can be computed:

$$h_t = o_t \tanh(c_t). \tag{12}$$

Finally, an RNN with LSTM architecture is implemented by replacing the recurrent hidden layer with an LSTM cell. In an LSTM cell, the input gate, forget gate, and output gate solve the vanishing and exploding gradient problems observed in RNN.

2.3 Model Predictive Control

MPC is a multivariable control algorithm that uses an internal dynamic model of the process and the history of past control moves to compute optimal control actions. In simple terms, it involves solving the following optimization problem, at each sample time, over a predetermined prediction horizon N,

$$\min_{\{u_1,\cdots,u_N\}} \sum_{t=1}^{N} w_{y_t}(y_{sp} - y_t)^2 + \sum_{t=1}^{N} w_{u_t}\Delta u_t^2 \tag{13}$$

subject to process and model constraints. In the objective function above, y_t is the process output at time step t, y_{sp} is the output setpoint, u_t is the control action at time step t, w_{y_t} is the weighting coefficient on controller error, and

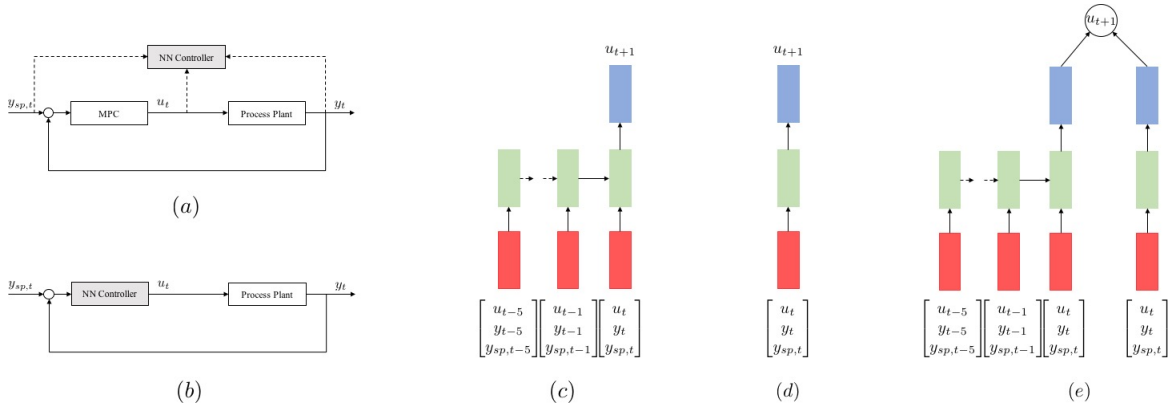

Fig. 2. (a) Training setup for an NN controller. (b) Implementation of an NN controller. (c–e) Architectures of an LSTM-only controller, NN-only controller, and LSTM supported NN (LSTMSNN) controller, respectively. Controllers are represented using standard NN block diagrams: the red blocks indicate the input layer, the green blocks represent the hidden layer, and the blue blocks represent the output layer.

w_{u_t} is weighting coefficient on u_t. In a receding horizon implementation, only u_1 is applied on the system and the full N-step optimization problem is re-solved in at every time step. In the following section, a novel combination of LSTM and NN is shown to replicate the dynamic behaviour of the MPC.

3. NN CONTROLLER

As discussed in Section 1, to address the computational issues of implementing an MPC for complex systems, we propose a novel NN controller architecture. The idea is straightforward: we first design the *best-in-class* MPC for our process, and then simulate it to generate pairs of process outputs and control actions. Next, the generated input-output data is used to train an NN controller. Once the NN controller is trained, it is deployed for real-time process control. A schematic of training and deploying an NN controller is shown in Figures 2(a) and (b), respectively.

We consider three separate architectures for training the NN controller: LSTM-only, NN-only, and LSTM-supported-NN (LSTMSNN). All the three models are trained using the data generated from an MPC.

The proposed LSTM-only controller is a 2-layer RNN of length 5, with each hidden layer containing several hidden units, and each hidden unit represented by an LSTM unit. For simulation, we use LSTM with 2 hidden layers with 20 hidden units in the first layer and 50 units in the second layer. The feed forward neural network has two layers with 20 and 40 units in the first and second layer respectively. An illustration of the proposed LSTM-only model is given in Figure 2(c). As shown in Figure 2(c), the inputs to the LSTM-only controller at each time step are the MPC control actions, plant outputs and set-points, denoted generically by $u \equiv [u_k, y_k, y_{sp,k}]^{\mathrm{T}}$, where $u \in \mathbb{R}^{n_u+2n_y}$ and $k = t-5, \ldots, t$. The network output is the control action taken at the next time step, denoted by u_{t+1} in Figure 2(c). Observe that the output is defined only at the terminal time, as opposed to at each time. The LSTM-only controller captures the dependency of u_{t+1} on the past inputs, outputs and set-points. Physically, the LSTM-only controller learns how the MPC control actions are generated based on the past control actions and the process outputs. Note that although the current inputs, outputs and set-points are included in the LSTM-only controller their influence may be limited due to the forget, input and output gates in the LSTM unit that control the amount of information that passes through the network.

To assign higher weights to the current inputs, outputs and set-points, we consider a NN-only control architecture shown in Figure 2(d). The proposed NN-only model is a 3 layer NN, with each layer containing several hidden units. The input to the NN-only model is $u \equiv [u_t, y_t, y_{sp,t}]^{\mathrm{T}}$, and the output is u_{t+1}. A NN-only model captures the MPC response based on current control actions and current outputs by discarding the past. In other words, a NN-only controller is a memory-less controller.

The third NN controller – LSTMSNN combines the benefits of an LSTM-only controller (which takes past control actions and outputs into consideration) and an NN-only controller (which takes current control actions and outputs into consideration) in a unified architecture. An LSTMSNN controller output is a weighted combination of an LSTM-only controller output and the NN-only controller output. This allows an LSTMSNN controller to effectively learn an optimal control action given past and current inputs, outputs and set-points. A schematic of the proposed LSTMSNN controller is shown in Figure 2(e).

4. PROCESS SETUP AND DATA COLLECTION

In this section, we demonstrate the efficacy of the proposed NN controller to control the moisture content of the sheets produced by a paper-making machine. The variable to be controlled is the moisture content and the manipulated variable is the steam flow rate. The following transfer function describes the relation between the steam flow and the moisture content:

$$G(z) = \frac{0.05z^{-4}}{1 - 0.6z^{-1}}. \tag{14}$$

A standard MPC architecture to control the drying process represented in (14) is designed in `MATLAB`. The NN controller is trained using `Tensorflow` (Abadi et al., 2016), a deep-learning framework. In `Tensorflow`, a NN is typically represented as graph, and the network gradients

are computed using automatic differentiation. To speed up the implementation of the NN controller, we use a GPU (`NVIDIA Geforce 960M`) with `CUDA`. The LSTMSNN model takes around 0.2 seconds to return a control action on this GPU. Finally, both the MPC and the process model are simulated in `MATLAB` to generate artificial training data for different initial process conditions and different reference trajectories.

All the three versions of the proposed NN controller – LSTM-only, NN-only and LSTMSNN – require three variables for training: the reference trajectory, process output and MPC control actions.

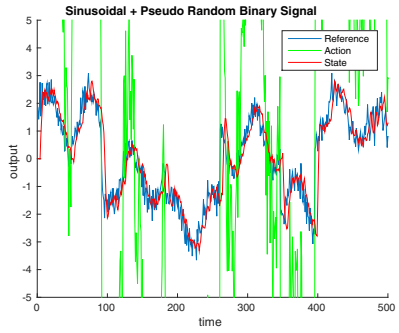

Fig. 3. 500 time steps of a mixture of binary, sinusoidal and combined sequences used for training. The reference signal is represented by the blue trajectory, the process output by the red trajectory and the LSTMSNN control action by the green trajectory.

The process is simulated for 1000 time steps. A small Gaussian noise is added to the process output to account for measurement noise and other uncertainties. To control the process, we also need to define a reference signal. We choose a pseudo-random binary reference signal in the intervals $[-3.0, -0.2]$ and $[0.2, 3.0]$. We also select a sine reference signal with period ranging in the interval $[10, 1000]$. Note that these intervals are defined for deviation process variables. Finally, note that a pseudo-random and sine reference signals ensure that the process is sufficiently excited in all frequencies. This also ensures that the data generated from the simulation is sufficiently informative for training a NN controller. A subset of data used for training is shown in Fig. 3.

5. RESULTS AND DISCUSSION

Table 1 compares the designed LSTM-only, NN-only, and LSTMSNN controllers using the mean square error (MSE) and offset error (OE). Observe that the LSTMSNN controller outperforms the LSTM-only and the NN-only controller, both in terms of MSE and OE, for all the chosen set-points. This is because unlike the LSTM-only or the NN-only controller, the LSTMSNN controller learns the optimization and prediction steps of an MPC within the weights of the network by studying the past and current control actions, plant outputs and set-points. Also, the LSTM only model does not converge because the problem formulation of LSTM is posed in a way to predict future control action given the past, whereas the LSMSNN model converged because the LSTM part does the time series prediction and the neural network part does the regression step given the current output and required target output.

Figures 4(a) and (b) compare the performance of the LSTMSNN and NN-only controllers for a (constant) set-point $y_{sp,t} = 5$. It is clear that the NN-only controller results in a significant target offset. Figures (4(c) and (d) compares the LSTMSNN and NN-only controllers for a set-point $y_{sp,t} = 10$, which is outside of the training data set. Again, the LSTMSNN controller is able to track the target set-point, demonstrating the better generalization of the controller over the larger process state space.

Figures 4(e)–(h) show the performance of the LSTMSNN-controller in tracking time-varying reference signals. Evidently the LSTMSNN controller can effectively track complex set-points. Finally, Figure 5 shows the tracking performance of the LSTMSNN controller under different process initial conditions.

These findings clearly establish the robustness and generalization capabilities of the designed LSTMSNN controller in effectively controlling the process under different initial conditions and for different reference signals. Also for simulation purpose we use pseudo random signals to collect data from MPC, practically any other signals like smooth step signals can be used to collect data from MPC.

Table 1: Performance comparison between methods

Model	Setpoint	MSE	OE
LSTMSNN	2	0.06	0.01
NN-only	2	0.07	0.23
LSTM-only	2	5.56	Did not converge
LSTMSNN	5	0.078	0.01
NN-only	5	0.53	0.7
LSTM-only	5	62.42	Did not converge
LSTMSNN	10	0.01	0.01
NN-only	10	2.21	1.3
LSTM-only	10	167.78	Did not converge

6. CONCLUSIONS

We have developed a novel deep NN controller that learns the optimal policy implemented by an MPC. The proposed NN controller eliminates the need to solve in real-time a complex optimization problem, state estimation problem and a prediction problem. This makes the NN controller (with GPU implementation) exceptionally fast and easy to implement compared to the traditional MPC. The performance of the designed NN controller was illustrated on an example problem from the paper industry.

ACKNOWLEDGEMENTS

We would like to thank Tingke Shen, Zilun Peng of University of British Columbia for helpful discussions.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard,

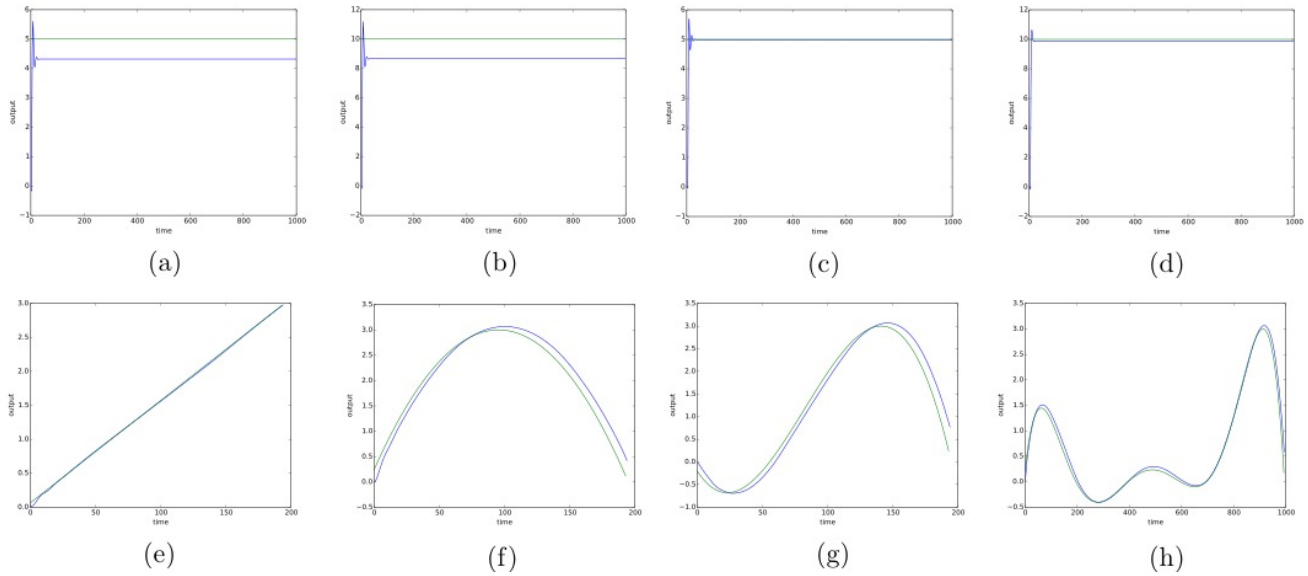

Fig. 4. (a–b) Performance of a NN-only controller for $y_{sp,t} = 5$ and $y_{sp,t} = 10$, respectively. (c–d) Performance of an LSTMSNN controller for $y_{sp,t} = 5$ and $y_{sp,t} = 10$, respectively. (e–h) Performance of an LSTMSNN controller for linear, quadratic, cubic and polynomial tracking targets, respectively. Set-points are shown in green and process outputs in blue.

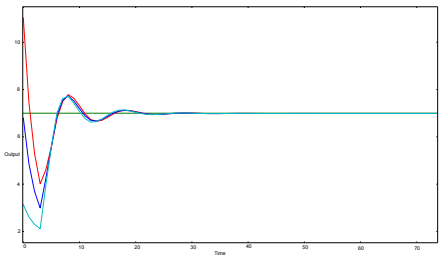

Fig. 5. Tracking performance of the LSTMSNN-controller for different initial process conditions. The set-point is shown in green.

M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, 265–283.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2), 157–166.

Biegler, L.T. (2000). Efficient solution of dynamic optimization and nmpc problems. In *Nonlinear model predictive control*, 219–243. Springer.

Biegler, L.T. (2007). An overview of simultaneous strategies for dynamic optimization. *Chemical Engineering and Processing: Process Intensification*, 46(11), 1043–1053.

Ellis, M., Durand, H., and Christofides, P.D. (2014). A tutorial review of economic model predictive control methods. *Journal of Process Control*, 24(8), 1156–1178.

Findeisen, R., Allgöwer, F., and Biegler, L.T. (2007). *Assessment and future directions of nonlinear model predictive control*, volume 358. Springer.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735–1780.

Jithesh, V., Sagayaraj, M.J., and Srinivasa, K. (2017). LSTM recurrent neural networks for high resolution range profile based radar target classification. In *Computational Intelligence & Communication Technology (CICT), 2017 3rd International Conference on*, 1–6. IEEE.

Kittisupakorn, P., Thitiyasook, P., Hussain, M., and Daosud, W. (2009). Neural network based model predictive control for a steel pickling process. *Journal of Process Control*, 19(4), 579–590.

Kothare, M.V., Balakrishnan, V., and Morari, M. (1996). Robust constrained model predictive control using linear matrix inequalities. *Automatica*, 32(10), 1361–1379.

Mayne, D.Q. (2014). Model predictive control: Recent developments and future promise. *Automatica*, 50(12), 2967–2986.

Piche, S., Keeler, J.D., Martin, G., Boe, G., Johnson, D., and Gerules, M. (2000). Neural network based model predictive control. In *Advances in Neural Information Processing Systems*, 1029–1035. NIPS.

Pon Kumar, S.S., Gopaluni, B., and Loewen, P. (2017). Deep reinforcement learning approaches for process control. In *Advanced Control of Industrial Processes (AdCONIP), 2017 6th International Symposium on*, 201–206. IEEE.

Qin, S.J. and Badgwell, T.A. (2003). A survey of industrial model predictive control technology. *Control engineering practice*, 11(7), 733–764.

Wallace, M., Pon Kumar, S.S., and Mhaskar, P. (2016). Offset-free model predictive control with explicit performance specification. *Industrial and Engineering Chemistry Research*, 55(4), 995–1003.

Wan, Z. and Kothare, M.V. (2003). An efficient off-line formulation of robust model predictive control using linear matrix inequalities. *Automatica*, 39(5), 837–846.

Yu-Geng, X., De-Wei, L., and Shu, L. (2013). Model predictive control? Status and challenges. *Acta Automatica Sinica*, 39(3), 222–236.