Deep Reinforcement Learning Approaches for Process Control

by

Steven Spielberg Pon Kumar

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Chemical and Biological Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

December 2017

Abstract

The conventional and optimization based controllers have been used in process industries for more than two decades. The application of such controllers on complex systems could be computationally demanding and may require estimation of hidden states. They also require constant tuning, development of a mathematical model (first principle or empirical), design of control law which are tedious. Moreover, they are not adaptive in nature. On the other hand, in the recent years, there has been significant progress in the fields of computer vision and natural language processing that followed the success of deep learning. Human level control has been attained in games and physical tasks by combining deep learning with reinforcement learning. They were also able to learn the complex go game which has states more than number of atoms in the universe. Self-Driving cars, machine translation, speech recognition etc started to gain advantage of these powerful models. The approach to all of them involved problem formulation as a learning problem. Inspired by these applications, in this work we have posed process control problem as a learning problem to build controllers to address the limitations existing in current controllers.

Lay Summary

The conventional controllers like Proportional, Proportional Integral controllers, etc. requires mathematical expression of the process to be controlled, involves controller tuning. On the other hand, optimization based controllers like Model Predictive Control are slow for controlling complex systems especially with a large number of inputs and outputs because of the optimization step. Our work aims at addressing some of the drawbacks existing in conventional and optimization based process control approach. We have used an artificial intelligence approach to build a self-learning controller to tackle these limitations. The self-learning controller learns to control a process just by interacting with the process using artificial intelligence algorithm. Some of the advantages of this approach are, it does not require the mathematical expression of the process, does not involve controller tuning and it is fast since it does not have optimization step.

Preface

The thesis consists of two parts. The first part which is discussed in Chapter 3, discusses about the new neural network architecture that was designed to learn the complex behaviour of Model Predictive Control (MPC). It overcomes some of the challenges of MPC. The contributions of this part have been submitted in:

Referred Conference Publication:

Steven Spielberg Pon Kumar, Bhushan Gopaluni, Philip Loewen, *A deep learning architecture for predictive control*, Advanced Control of Chemical Processes, 2018 [submitted].

The contributions of this part have also been presented in:

Conference Presentation:

Steven Spielberg Pon Kumar, Bhushan Gopaluni, Philip Loewen,*A deep learning architecture for predictive control*, Canadian Chemical Engineering Conference, Alberta, 2017.

The second part of the work explained in Chapter 4, discusses the learning based approach to design a controller to be able to control the plant. The motivation for this approach is to solve some of the limitations present in the conventional and optimization based controllers. The contributions of this part have been published in:

Referred Conference Publication:

Steven Spielberg Pon Kumar, Bhushan Gopaluni and Philip D. Loewen, *Deep Reinforcement Learning Approaches for Process Control*, Advanced Control of Industrial Processes, Taiwan, 2017

The contributions of this part have also been presented in:

Presentation:

Steven Spielberg Pon Kumar, Bhushan Gopaluni, Philip Loewen, *Deep Reinforcement Learning Approaches for Process Control*, American Institute of Chemical Engineers Annual Meeting, San Francisco, USA, 2016.

Table of Contents

List of Tables

List of Figures

Acronyms

Acronyms	**Full Form**
DL	Deep Learning
DRL	Deep Reinforcement Learning
LSTM	Long Short Term Memory
LSTMSNN	Long Short Term Memory Supported Neural Network
MPC	Model Predictive Control
NN	Neural Network
P - Controller	Proportional - Controller
PI - Controller	Proportional Integral - Control
PID - Controller	Proportional Integral Dervative Controller Control
RL	Reinforcement Learning

Acknowledgements

It gives me an immense pleasure to express my deep sense of gratitude and thankfulness to Dr. Bhushan Gopaluni who through more than 700 emails, many meetings, phone calls and Skype conversations over two years, chiseled me from an a curious student to a competent researcher and developer. Dr. Bhushan's vision, passion, aspiration are infectious. Whenever I had a narrow idea he transformed those ideas into a long term broad visionary one and patiently repeated them and helped me until I got it. Whenever I sent him an unstructured draft he patiently gave me many feedback and corrections till I got a good one. He also gave me enough freedom in terms of approaching the research problem we have been working and in granting me permissions to take courses relevant to my research. He gave me opportunities to help me get relevant skills needed for the project. The two years at UBC was a lot of learning and it would not have been possible without Dr. Bhushan. His teachings have helped me not only during project period but will inspire me in future also.

Next, I am grately thankful to Dr. Philip Loewen on helping me throughout my masters project and for giving me the kind opportunity to pursue research in such an esteemed field (machine learning + process control). He was very encouraging and been a constant support to me. I am very thankful to his detailed comments on my writing and for his encouragement during my presentation. His comments will help me throughout my career.

I am highly grateful to Dr. Ryozo Nagumune for his time in reviewing my thesis and in giving me valuable feedback and corrections.

I would like to thank Dr. Mark Schmidt in helping me give necessary platform to improve my machine learning skills. I want to thank my department for being very responsive and supportive especially I want to express my gratitude to Marlene Chow. I am thankful to Dr. Peter Englezos (Head of Department Chemical and Biological Engineering), Dr. Elod Gyenge (Graduate Advisor), Rockne Hewko (Graduate and Post Doctoral Studies), Gina Abernathy for helping me throughout my masters.

I would like to thank Smriti Shyammal, McMaster University for being very supportive during my ups and downs. His intuitive recommendations, suggestions, guidance supported with case studies, thoughtful discussions and examples was one of the best things during my course of my masters degree.

I have to thank my lab mates Lee Rippon, Quigang Lee, Yiting, Hui who were available whenever I had clarifications. I would like to thank Mitacs for the financial support. I also want to express my thanks to Kevin Shen and Zilun Peng for helpful discussions. I am also indebted to Dhruv Gupta, University of Wisconsin Madison and Raj Ganesh for being a constant support.

Last but not the least I would like to thank my mother Packiavathy Geetha for all the struggles and hard work she put to raise me. Thanks for all her sacrifice to educate me. I also want to thank my father for being supportive to my family. A big thank you to my brother for being very supportive to me and to my family without my live presence back home.

Thanks to Ebenezer, Annie and Mohan for being supportive. Finally, I am always thankful to Lord Almighty for all the opportunities and success he has given me in life.

Dedication

To my mother Packiavathy Geetha.

Chapter 1

Introduction

1.1 Overview

One of the dreams of the field of process control is to enable controllers to understand the plant dynamics fully and then act optimally to control it. It could be challenging because online learning is relatively easy for human beings because of the evolution compared to machines.

The current approaches for control are either classic control approach or optimization based approach. These approaches do not take intelligent decisions but obey certain rules that was depicted to the controller by a control expert. The conventional approaches require extensive knowledge from an expert with relevant domain knowledge to transfer the knowledge to the controller via control law and other mathematical derivation. On the other hand, optimization based controllers like Model Predictive Controllers look ahead in the future and take action considering future errors but suffer from the fact that the optimization step takes time to return optimal control input, especially for complex high dimensional systems. The detailed drawback of the current approaches for process control is discussed in the limitation section below.

1.2 Limitations

There are many limitations with the current approaches to industrial process control. The classical controller requires very careful analysis of the process dynamics. It requires a model of the process either derived from first principles or empirical. If the system under consideration is complex then the first principle model requires advanced domain knowledge to build the model. In certain cases, the conventional approaches involves derivation of control law that meets desired criteria (either to control the plant to track the set point or operate in a way to be economical). It involves tuning of controllers with respect to the system to be controlled. Model maintenance is very difficult or rarely achieved. On the other hand, optimization based controllers look ahead into the future and take future errors into account. However, they suffer from the fact that computing optimal control actions via optimization tends to take time especially for non-linear high dimensional complex system. They cannot handle uncertainty about the true system dynamics and noise in observation data. One more burden of optimization based controller like MPC is the prediction of hidden states, since often those states are not readily available. MPC also demands to tune prediction horizon, control horizon, weights given to each output etc. As a prerequisite, MPC requires an accurate model of the system to start with. Also over the course of time the controller will not possess an updated model, hence the performance will start to deteriorate.

1.3 Motivation

Having given the above limitations can we build a learning controller that does not require 1) intensive tuning 2) plant dynamics 3)model maintenance, and 4) takes instant control actions? This work aims to develop such next generation intelligent controllers.

1.4 Challenges

It could be challenging to learn the entire trajectory of a plant's output along with the set point. Deploying these learning controllers in production could be risky, hence it should be made sure that the controller has been already trained offline before implementing it in the real plant. The action space and state space in process control application are continuous, hence function approximators are used with reinforcement learning to learn the continuous state space and action space. Since it has been found that deep neural networks serve as an effective function approximators and recently have found great success in image [1], speech [7] and language understanding [8], deep neural networks are used as common function approximators with reinforcement learning. However, to speed up the training of deep neural networks, it requires training them in the Graphical Processing Unit (GPU) to parallelize the computation. This could be tricky to train as care should be taken to optimize the memory allotted to perform computations in the GPU. The reinforcement learning based controller involves training two neural networks simultaneously, tuning those two neural networks for the first time to train the plant was challenging. The reason is the enormous number of options and space available as tunable parameters (hyper parameters). In addition, running simulation takes a lot of time, hence re tuning the hyper parameters to retrain the network was bit challenging.

1.5 Encouraging progress

In recent years, there has been significant progress in the fields of computer vision and natural language processing that followed the success of deep learning. The deep learning era started in 2012 after it won the image classification challenge in 2012, by a huge margin compared to other approaches [1]. In Machine translation conventional probabilistic approaches were replaced with deep neural networks [53]. From then on, self driving cars started to take the advantage of deep nets [54]. They use deep learning to learn the optimal control behaviour required to drive the car without the help of human. This is a form of a learning control problem. Human level control has also been attained in games [2] and physical tasks [3]. Computers were able to beat human beings in classical Atari games [2]. Deep neural nets have even learnt the complex game of Go [15]. This is again posed as a learning control problem. The Go game has 1.74×10^{172} states, which is more than the number of atoms in the universe. The question is can we gain advantage of those powerful models to build a learning controller to overcome the limitations of the current control approach? Our work aims to answer this question. To support using such powerful models, the number of calculations per second has been growing exponentially over the years.

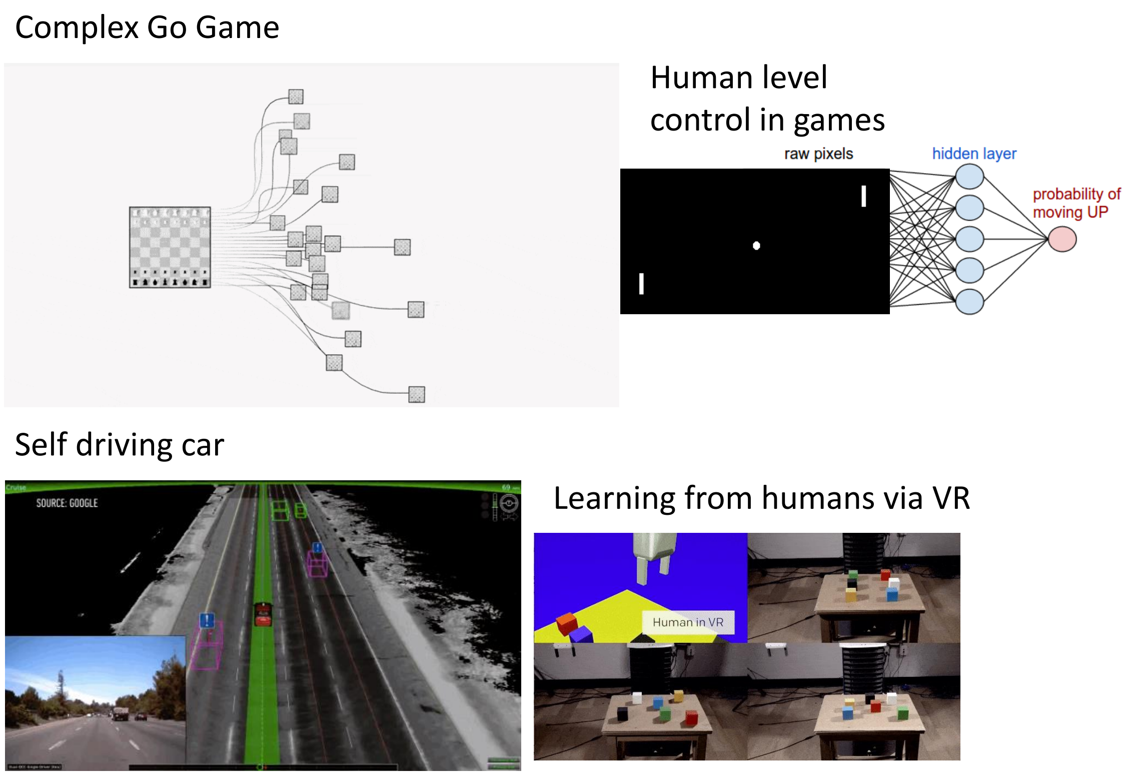


Figure 1.1: Success of Deep learning and motivation for learning based controller [43],[44],[45],[46]

1.6 Contributions and Outline

We present two principal contributions,

1) Optimization based control We propose a new deep learning architecture to learn the complete behaviour of MPC. Once the deep neural network is learnt, it can take action instantaneously, without having to estimate hidden states, with no prediction and optimization as these information will be learnt while training.

2) Classical Control We have developed a learning based controller that can learn plant dynamics and determine control actions by just interacting with the plant. These controllers does not require tuning and are adaptive in nature. Also, the learnt controller are fast in returning optimal control actions.

The thesis is organized as follows: Chapter II provides background information about machine learning, deep learning, MPC and Reinforcement Learning. Chapter III discusses our contribution to optimization based controller. Chapter IV discusses our work on learning based controller design using deep reinforcement learning. Chapter V presents conclusions and future directions of this work.

The work discussed in Chapter III is based on supervised learning with the data from MPC to learn control behaviour, whereas the work discussed in Chapter IV is based on reinforcement learning to learn the control behaviour. The supervised learning with MPC is preferred, if MPC has to be performed on a complex system, because optimization step on complex system takes time and the optimization step is learnt offline with the data from MPC in the supervised approach. The reinforcement learning approach is preferred, 1) when it is tedious to develop or derive plant model 2) a controller that is susceptible to change in plant model is needed 3) the control behaviour are learnt using function approximators and the learnt control policy (mapping from states to actions) with function approximators are fast.

Chapter 2

Background

The thesis has two parts 1) Using deep learning to learn the control behaviour of Model Predictive Control, 2) Using deep learning and reinforcement learning to learn the optimal behaviour online just by interacting with the plant. Hence, this chapter discusses the background on machine learning, neural networks, Model Predictive Control (MPC) and deep reinforcement learning. For detailed information about these topics we recommend the Deep learning book from Goodfellow et al. [16] and the Reinforcement Learning book from Sutton and Barto [17].

2.1 Machine Learning

Let us discuss machine learning with an example. Consider the following data set given in table 2.1 showing Time taken to move a box 50m vs mass of the box.

Table 2.1: Sample input output data

Box Mass (kg)	Time taken to move 50 m (seconds)
1	2.08
2	4.95
3	5.63
4	7.16
5	11.76
6	14.39
7	17.03
8	16.06
9	18.46
10	20

Now let us visualize this data as shown in figure 2.1.

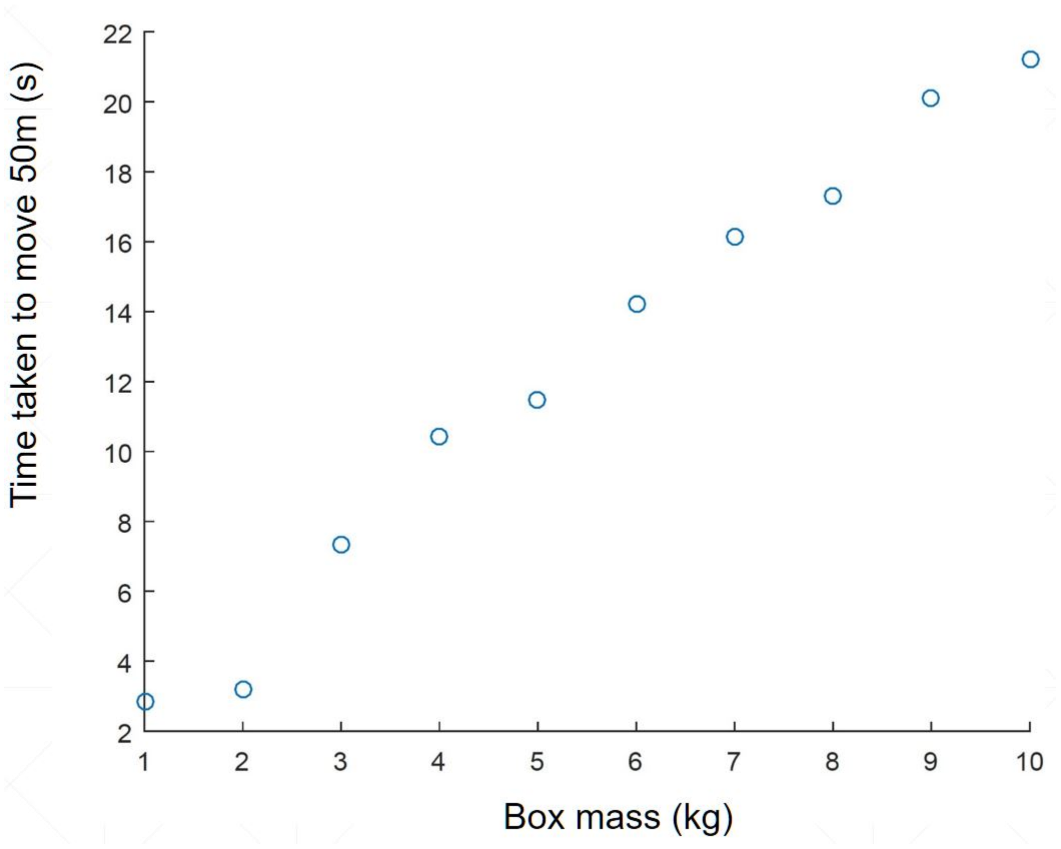


Figure 2.1: Plot of data given in table 2.1

The data is given only for 10 specific boxes. What if we want to find the time taken to move a 5.5 kg box? In this case we have to learn from the data. Learning the data essentially means learning the parameters or weights of a machine learning model. Since the data visualized in figure 2.1 looks linear, a linear model is enough to learn the data.

$$y = \theta_1 x + \theta_2 \tag{2.1}$$

where θ_1 and θ_2 are parameters or weights of the equation. Equation 2.1 is known as linear regression machine learning model. The parameters θ_1 and

θ_2 are learnt using optimization techniques which will be discussed later in this chapter. Once θ_1 and θ_2 are learnt the data can be plotted along with the machine learning model, in this case a line as follows:

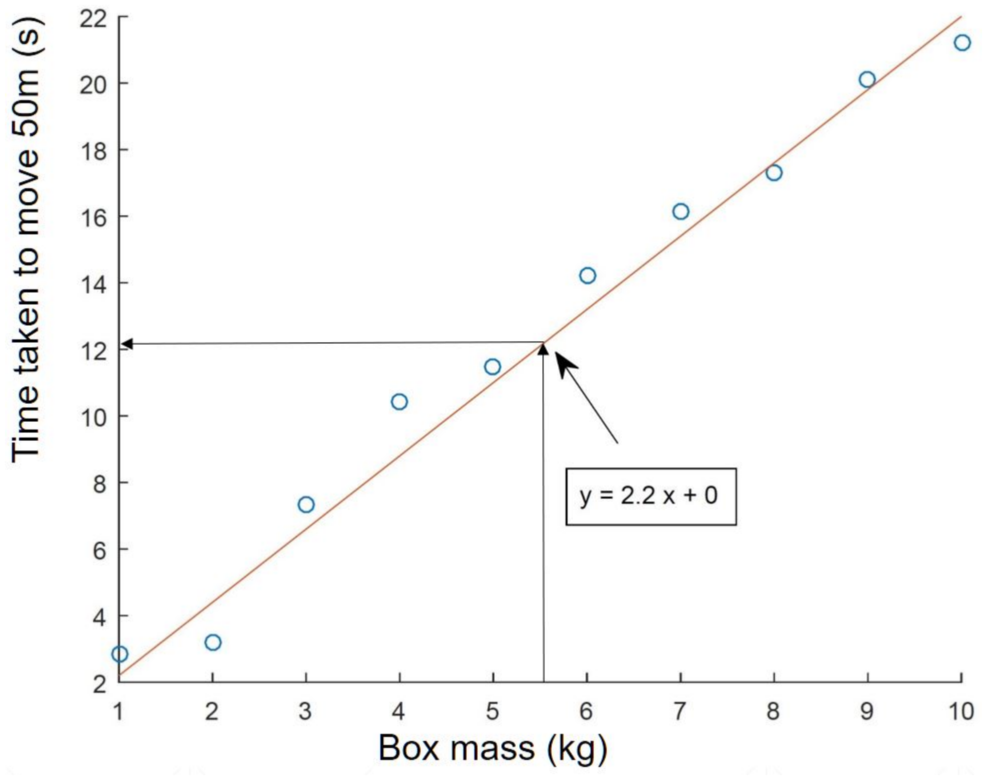


Figure 2.2: Plot of data along with learnt line using optimization technique

Now if we know the line that is learnt using the data we had, we can answer questions related to the data set even for box weight data we do not have. θ_1 is the slope of the line and θ_2 is the intercept of the model. For instance, for the above data set and for linear regression model, the parameters θ_1 and θ_2 learnt using optimization techniques are $\theta_1 = 2.2$ and $\theta_2 = 0$. Then the time taken to move a 5.5 kg box 50 m would be $2.2{\times}5.5{+}0 =$

12.1s. For learning the parameters of this linear model, we are supervising our algorithm with input and corresponding output data, hence it is called supervised learning. We will discuss this further in the next section.

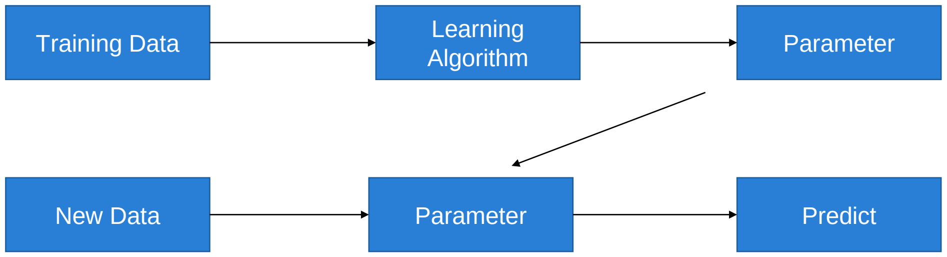


Figure 2.3: Machine learning framework.

In general the machine learning framework can be summarized with the flow diagram in Figure 2.3. First we have training data and we use optimization technique to learn the parameters. The learnt parameters can be used with the new data to predict the output of the new data.

2.2 Supervised Learning

We saw a simple example of machine learning, now let us go through supervised learning in detail. Many practical problems can be formulated as a mapping $f : X \mapsto Y$, where X is an input space and Y is an output space. For instance, in the above example, the input space is mass and the output space is time. In most cases it is not easy to come up with the mapping function f manually. The supervised learning framework provides an approach that takes advantage of the fact that it is often relatively easy to obtain sample pairs $(x, y) \in X \times Y$ such that $y = f(x)$. In the above example data set, this would correspond to collecting a data set of time taken to move 50m of box considered in the input space annotated by humans.

Supervised Learning Objective Precisely, we have a ”training data set” of n examples $(x_1, y_1), ...(x_n, y_n)$ made up of independent and identically distributed (i.i.d.) samples from a data generating distribution D; i.e., $(x_i, y_i) \sim D$ for all i. We frame learning the mapping $f : X \mapsto Y$ by searching over a set of candidate functions and finding the one that is most consistent with the training examples. D is the distribution from which the data is being generated and we are interested in finding the function $f*$ (from set of functions f) that can better approximate the data generation distribution D. Concretely, we consider particular set of functions $\mathcal{F}$ and choose a loss function which is a scalar $L(x, \hat{y}, y)$ that expresses the disagreement between a predicted label $\hat{y}_i = f(x_i)$ for some $f \in \mathcal{F}$ that meets the condition below:

$$f* = \arg \underset{f \in \mathcal{F}}{min} E_{(x,y)} \sim_D L(f(x), y) \tag{2.2}$$

From a big picture, we are interested in a function $f*$ that minimizes the expected loss over the distribution D from which the data is generated. Once the function is determined, we can only have this function and use this function with the new training data set to do the mapping from X to Y. The initial training data can be discarded.

Unfortunately, the above optimization objective cannot be solved easily because we do not have all possible samples (or elements) of D and hence the expectation cannot be evaluated analytically without making assumptions about the form of D, L or f. If we assume that the available samples from

D are i.i.d. (Independent and Identically Distributed) we can rewrite and approximate the expectation given in equation (2.2) by averaging the loss over the training data available as follows:

$$f* \approx \arg \min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} L(x_i, f(x_i), y_i) \tag{2.3}$$

Even though we approximate the expectation with data available in training, we believe that it is a good approximation to equation (2.2), especially if we have a large number of data and it serves as a good proxy to the objective to equation (2.2).

Overfitting Sometimes, there are cases where the equation given in (2.2) learns too much about the training data that it starts to over fit without learning the general pattern of the data. This scenario is called overfitting.

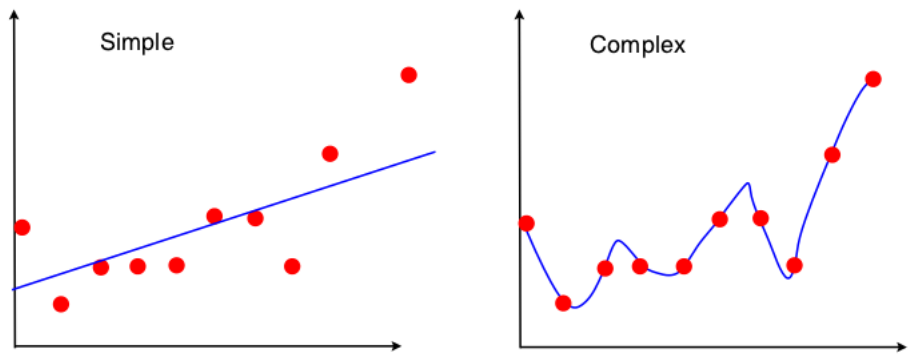


Figure 2.4: left: Learnt linear function on data (red dot); right: example of overfitting [47]

Figure 2.4 is used to discuss about overfitting. The red dots represent the data points and the blue line the function $f*$ learnt from the data using optimization. The trend of the data looks linear, hence a linear function as shown on the left hand side is enough to learn the data, because the deviation of data points away the blue line could be considered as noise. On the other hand, in the right hand side, the function (blue line) interpolates all the points without learning the generalized pattern of the underlying data. The right hand side figure is an example of function over fitting the data.

Regularization Consider a function f that maps x_i to y_i in the training data, but for data other than training data it returns zero. This would be the result of equation (2.3) when the minimum value is attained at $y = \hat{y}$. Technically, we expect higher loss for all points in D that are not in the training data. The reason to the problem is the optimization objective tries to look for a weights where y and $\hat{y}$ are almost equal so as to be closer. To make the objective function generalize well the training data, an additional term R, called a regularization term, is added to the objective as follows:

$$f* \approx \arg \underset{f \in \mathcal{F}}{min} \frac{1}{n} \sum_{i=1}^{n} L(f(x_i), y_i) + R(f) \tag{2.4}$$

where R is a function which is scalar valued. It adds a penalty on different weights of the model to reduce the freedom of the model. Hence, the model will be less likely to fit the noise of the training data and will improve the generalization abilities of the model. Two common forms of regularization R, used are,

- L^1 regularization (also called Lasso)
- L^2 regularization (also called Ridge)

The L^1 regularization shrinks some parameters to zero. It is called L^1 regularization because the regularization term is the sum of absolute values (1 Norm) of weights/parameters of the model available. Hence some variables will not play any role in the model.

The L^2 regularization forces the parameters to be relatively small, if the penalization is bigger and vice versa. It is called L^2 regularization because the regularization term $R(f)$ is the 2 norm of the weights available. Hence, when the overall objective function along with regularization term is minimized, it shrinks the parameters/weights of the model.

Linear Regression Consider a 2 dimensional input data set of 100 samples represented as ($X = \mathbb{R}^2$) each corresponding to a scalar value of output ($Y = \mathbb{R}$). Since it is linear regression, the function $\mathcal{F}$ we consider is the set of linear functions that maps from X to Y (i.e., $\mathcal{F} = w^T x + b | w \in \mathbb{R}^2, b \in \mathbb{R}$). Since, our input has 2 dimensions (2 variables), 2 weights are needed to learn the model apart from one more weight which is required as a bias term. Hence, our hypothesis space has 3 parameters (w_1, w_2, b), where $w = [w_1, w_2]$

Now the next step is to learn the weights w_1, w_2 and b. Hence, a loss function has to be constructed. For regression, the commonly used loss function is the squared loss (squared difference between the predicted value and the target value of the training data), $L(\hat{y}, y) = (\hat{y} - y)^2$. As we saw earlier, we use regularization apart from the regular term given by, $R(w, b) = \lambda(w_1^2 + w_2^2)$. The regularization term favours parameters that are relatively small. Combining everything together, the optimization objective is given as,

$$\min_{w_1,w_2,b} \underbrace{\Big[\frac{1}{n}\sum_{i=1}^{n}(w^T x_i + b - y_i)^2\Big]}_{\text{fit the training data}} + \underbrace{\lambda(w_1^2 + w_2^2)}_{\text{regularization}} \tag{2.5}$$

It is common with models of the form $y_i = wx_i + b$ to regularize only weights w, leaving bias b free to be as large or small as it wants, with no penalty. The variance penalty for leaving the bias (intercept) unregularized is likely to be smaller than the rewards we reap it in terms of decreased bias, since most linear models do have some nonzero intercept. Also, the bias term does not interact multiplicatively with inputs; they merely allow the model to offset itself away from the origin. The bias term typically requires less data to fit accurately than the weights. Each weight specifies how two variables interact, and requires observing both variables in a variety of conditions to fit well. Each bias controls only a single variable. This means that we don not induce too much variance by leaving the biases unregularized. Regularizing, the bias term can introduce a significant amount of underfitting.

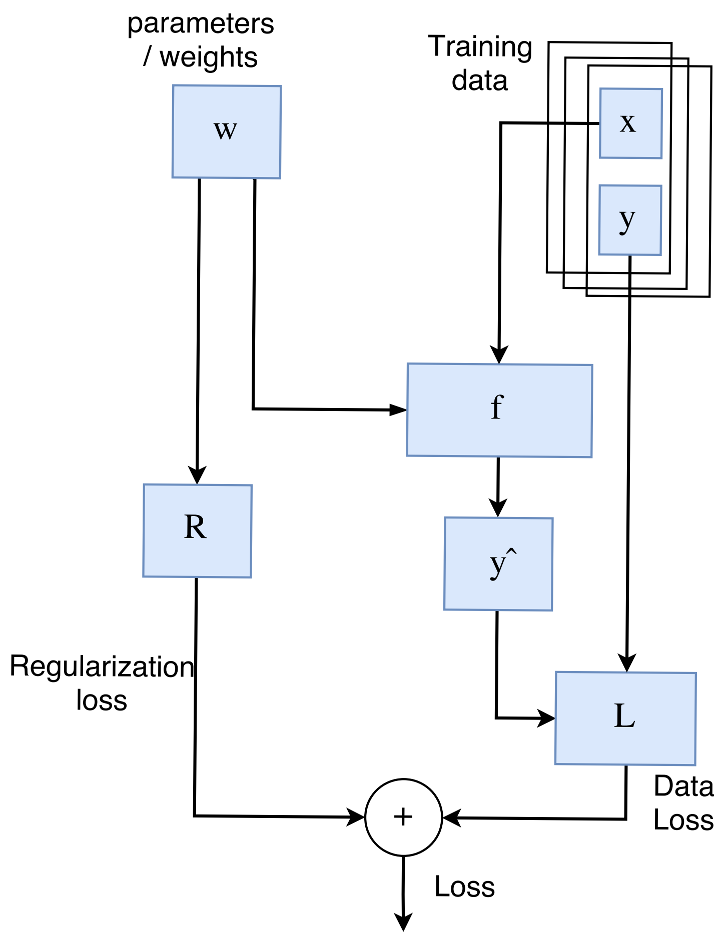


Figure 2.5: Data flow diagram of supervised learning framework in machine learning

Figure 2.5 shows the data flow diagram of supervised learning framework. The x is the training input and y is the corresponding label/ output of the training data. We have three decisions to make 1) The function f, which is a mapping from input x to output y, (a Neural Network in this dissertation), 2) The Loss function L, that is the squared error difference between predicted output from the machine learning output and the actual output (this dissertation has problems formulated only as regression, hence squared error is common throughout), 3) The regularization function R, since this dissertation only has regression formulation, the L^2 regularization is used. Once the predicted output with respect to weights and machine learning model is computed, the Data loss is calculated and then finally the overall loss is the summation of data loss and regularization loss.

Linear Regression in neuron representation We described linear regression in the previous section. Since we will deal with neural networks later, in this chapter, the linear regression can be represented in the diagrammatic notation popular for general neurons as follows:

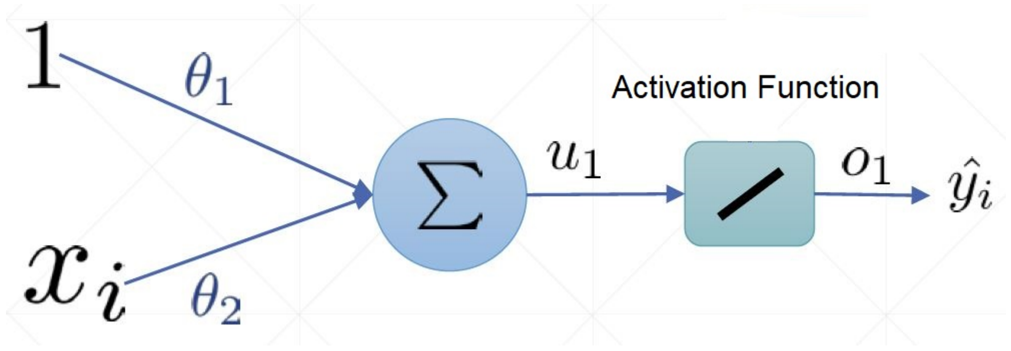


Figure 2.6: Neural Netowk representation for Linear regression on 1 dimensional data set.

Figure 2.6 shows the neural network representation for linear regression. In this case, the linear regression model is a weighted combination of inputs and weights. The θ_1 and θ_2 are weights or parameters here. The activation function in this case represented as slanting line is just the identity function ($o_1 = u_1$). More advanced activation functions will be used later in this chapter in neural networks.

Logistic Regression in neuron representation Let us consider linear regression for a binary classification task with labels 1 and 0. The drawback of linear regression is the output is not bounded between 0 and 1. This is addressed in logistic regression, which is a simple non-linear model for performing classification tasks. The logistic regression in neuron representation is given as follows:

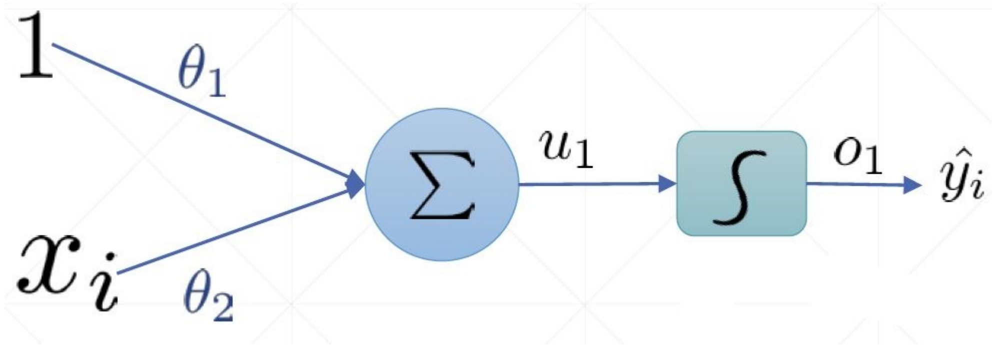


Figure 2.7: Logistic regression on 1 dimensional input data set.

Figure 2.7 show logistic regression in neural network representation. In this case, first the linear regression model which is a weighted combination of inputs and weights is used. The θ_1 and θ_2 are weights or parameters here. Since, the output is not bounded between 0 and 1, a sigmoid activation function given by $\sigma(x) = \frac{1}{1+e^{-x}}$ is used. This function squashes the output to be between 0 and 1. The predicted output $\hat{y}_i$ of logistic regression given in Figure 2.7 is,

$$\hat{y}_i = \frac{1}{1 + e^{-(\theta_1 + \theta_2 \times x_i)}} \tag{2.6}$$

The objective is to find the parameter θ that minimize the loss function given by,

$$L(x, \hat{y}, y) = -\sum_{k=1}^{K} y_k \log \hat{y_k} \tag{2.7}$$

The equality in equation 2.7 is the definition of cross-entropy between two distributions $H(p,q) = -\sum_x p(x) \log q(x)$. Since we interpret the output of logistic regression as containing probabilities of two classes (0 and 1), we see that we are effectively minimizing the negative log probability of the correct class. This is also consistent with a probabilistic interpretation of this loss as maximizing the log likelihood of the class y conditioned on the input x.

We saw linear regression and logistic regression. Now the goal is to learn all the components of the parameter vector θ using optimization techniques. Let us take a digression to discuss the optimization techniques used in machine learning.

2.3 Optimization

In linear regression we saw how the vector θ can be learnt from the data by minimizing the squared error between predicted output and actual labelled output. Specifically, $\theta^* = \arg\ \min_\theta g(\theta)$, where θ is the parameters to be learnt (usually in vector) and g is the sum of data loss and regularization loss. Let us see some of the gradient based techniques used to find the optimal parameters θ.

Derivative free optimization The simplest technique that one could think of is to evaluate $g(\theta)$ for various θ. One approach to this problem is the 'trial and error' approach. We can draw a number of θ at random from some distribution. For each θ drawn we evaluate $g(\theta)$, and then simply pick the one that minimizes g. Another method is to give perturbations to θ and continuously monitor the loss $g(\theta)$ (error in predicted and actual output). However, the trial and error approach and perturbing weights approach does not scale, as the neural networks discussed in this dissertation will have hundreds of thousands of parameters.

First order methods Instead of searching for the optimal parameters of the machine learning model in a brute force manner, we can make some assumptions about g to compute the parameters efficiently. If we assume g to be a differentiable function then, since our loss function is convex (for 1 parameter, its graph is a parabola), the function will be minimum at the point where slope of loss function with respect to parameter is zero. In other words, the slope or gradient ($\nabla_\theta g$) at minimum would be zero. The vector $\nabla_\theta g$ is composed of first derivatives of g. This is the reason why it is called a first order methods?.

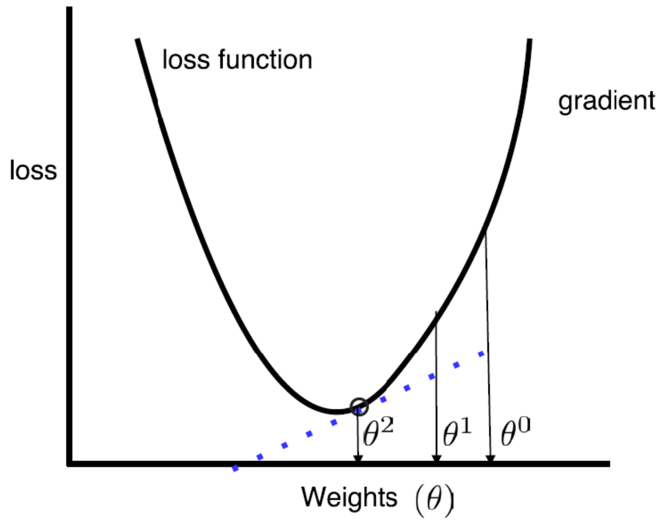


Figure 2.8: Loss function vs parameter.

We can use the information from the gradient to efficiently find the point of minimum. In short if we move away from the direction of gradient then we can approach the point of minimum. From figure 2.8, we can see that at θ^0 (i.e., at the point of initialization) the gradient direction is positive (positive slope). Since the slope is positive, we have to take a step (given by ϵ, the so called learning rate) in the opposite direction i.e., negative direction, resulting in the point θ^2. This process is continued until we reach the value of θ for which the loss function is minimum. This strategy is the base for the gradient descent algorithm. 1) Evaluate the gradient of the loss function with respect to parameters at the current point 2) Update the parameters of the model in the direction opposite to the sign of gradients. Gradient descent requires computing the gradient for all data samples. This may not be practically feasible, especially when the data set is large. Hence, we usually split the dataset into batches, compute the gradients, and then update the parameters.

If the size of the batch is 1 then it results in Stochastic Gradient Descent (SGD) which is summarized in Algorithm 1.

Algorithm 1 Stochastic Gradient descent

```
**Given** a starting point $\theta \in \mathbf{dom} g$
**Given** a step size $\epsilon \in R^+$
**repeat**
    1. Sample a mini batch of $m$ examples $(x_1, y_1), ..., (x_m, y_m)$ from the training data
    2. Estimate the gradient $\nabla_\theta g(\theta) \approx \nabla_\theta[\frac{1}{m}\sum_{i=1}^{m} L(f_\theta(x_i), y_i) + R(f_\theta)]$ with backpropogation
    3. Compute the update direction: $\delta\theta := -\epsilon\nabla_\theta g(\theta)$
    4. Update the parameter: $\delta := \delta + \delta\theta$
**until** convergence
```

An important parameter in the SGD is the learning rate ϵ which is related to the step size. If the value of ϵ is very small then the time taken to reach the minimum will be high. On the other hand if the value is too high the method won't converge and will oscillate around the minimum. Let us consider an example: to minimize the function, $g(\theta) = \theta^2$. The gradient of which is $g'(\theta) = 2\theta$. Let us initialize with $\theta = 1$. If $\epsilon > 1$ it will cause the gradient descent update to oscillate and diverge to infinity. On the other hand if $\epsilon = 1$, it will make the algorithm to oscillate between $\theta = 1$ and $\theta = -1$ infinitely. The third case being a good learning rate, $0 < \epsilon < 1$ will take the value of θ to zero, which is the point of minimum. It

Practically, a good rule of thumb is to pick a learning rate that makes the algorithm to diverge (e.g. 0.1). Having noted this value, set the learning rate to be slightly smaller than this amount (e.g. 0.05) as a safety margin. Gradually decaying the learning rate over time is a good practice. This can be achieved by multiplying the learning rate by a small factor (e.g. 0.1) once every few iterations to lower the learning rate. The convergence rates of this heuristic is discussed in [55] .Polyak averaging also provides consistent averaging [18]. In Polyak averaging the average parameter vector, $\bar{\theta}$ (e.g. $\bar{\theta} = 0.999\bar{\theta} + 0.001\theta$) is computed after every parameter update. At test time it uses $\bar{\theta}$.

Other optimization techniques. Usually the performance of gradient descent techniques can be improved by modifying the update step which is line number 3 in Algorithm 1. The first popular improvement that was made to gradient descent is the momentum update. It is framed in a way to progress along slow towards the direction of the gradient consistently. If the gradient is denoted as $\nabla_\theta g(\theta)$ for the gradient vector, the update step $\delta\theta$ is found by updating an additional step with the variable $v := \alpha v + g$ (initialized at zero). The update step now becomes $\delta\theta := -\epsilon v$. The variable v encourages the gradient update to proceed in a consistent (not identical) direction because it contains an exponentially-decaying sum of previous gradient directions. It comes with inspiration from physics, where the gradient is interpreted as the force F on a particle with position θ, hence this increments position directly, which is supported by Newton's second law $F = \frac{dp}{dt}$, where the mass m is assumed to be constant and $p = mv$ is the momentum term. The momentum update technique with a running estimate of the mean (first moment). Based on this, many similar methods of increased efficiency have been proposed.

- Instead of the intermediate variable v as in momentum step, the intermediate variable may have the update rule, $r := r + g \odot g$ of sum of squared gradients (where $\odot$ is element wise multiplication instead of vector or matrix multiplication). The update is given as follows: $\delta\theta := -\frac{\epsilon}{\gamma+\sqrt{r}} \odot g$, where γ is added to prevent the division by zero, usually the value of γ is set to 10^{-5}. This update is known as Adagrad update [19]

- Instead of running mean first moment if second moment which variance is used as given by: $r := \rho r + (1-\rho) g \odot g$, where ρ is usually set to 0.99. The update is known as RMSProp update [20]

- If the update step of Adagrad and RMSProp is combined, which is the estimates of both first and second running moments, the update is called the Adam update [21]. As in Adagrad and RMSProp, γ is usually set to 10^{-5} and ρ is usually set to 0.99. This update takes us to the optimum point with less oscillation. Throughout the dissertation during training the Adam method is employed for gradient descent update.

Cross-validation. We saw several other parameters called 'hyper parameters' apart from the regular parameters or weights we wanted to learn. Some of the hyper parameters are the learning rate ϵ, the regularization weights λ, the number of hidden units in neural network (see below) etc. Usually to determine good parameters, the available data set is split into training and validation samples. Once the model is trained on training set, it will be tested on validation set to see how well it performs. Usually this is done by comparing the predicted value of validation sample to true value of validation sample output. One familiar cross validation technique is the k-fold cross validation. Here the data are split into k subsets and then each subset will act as a validation set and finally the mean of all validation errors will be computed to test the efficiency of the model. Eg., 5-fold cross-validation will divide the data into 5 folds. Fig 2.9 shows visualization of k-fold cross validation.

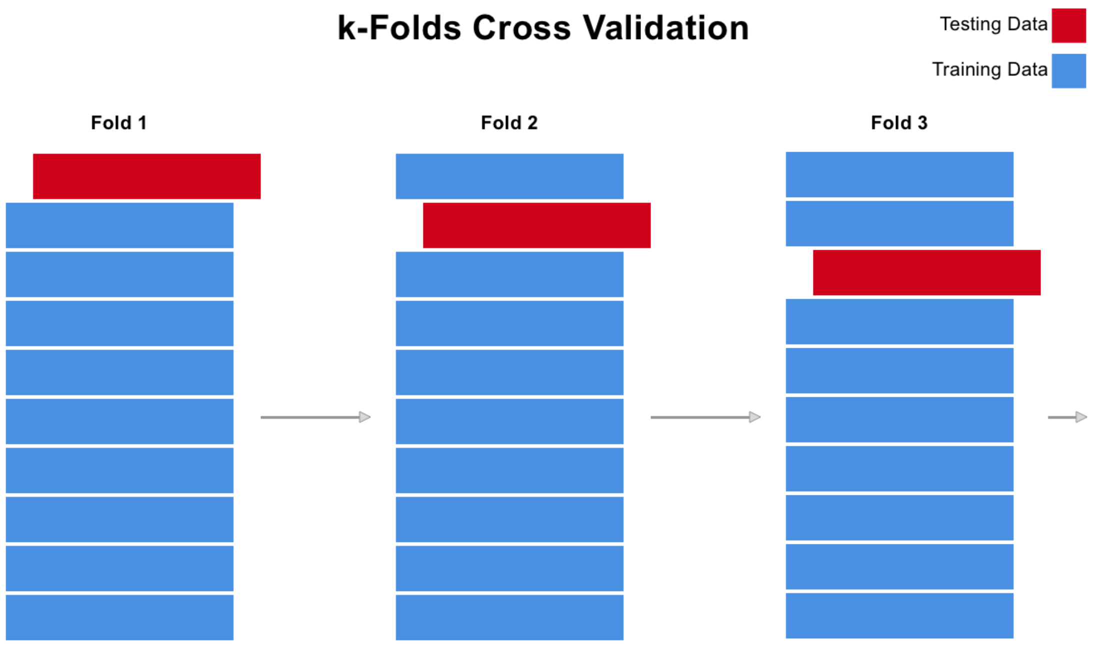


Figure 2.9: Figure shows how training and validation set are chosen for k-fold cross validation. The folds get extended to the right till the last fold is reached. [48]

Supervised Learning - Summary The supervised learning approach supervises the learning algorithm with known data. We are given a data set of n samples $(x_1, y_1), ..., (x_n, y_n)$ where $(x_i, y_i) \in X \times Y$, and we want to find the three components which are listed below:

1. The family of mapping functions $\mathcal{F}$, where each $f \in \mathcal{F}$ maps X to Y, it could be linear regression, logistic regression, neural networks etc.

2. The Loss function $L(x, \hat{y}, y)$ that evaluates the error between a true label y and a predicted label $\hat{y} = f(x)$. Loss functions vary slightly for

regression and classification based problems.

3. The regularization loss function $R(f)$, which is scalar, to help generalize the data better.

Since the dissertation focuses on problems formulated using regression, usually L^2 regularization is used as a $R(f)$ regularization loss. Our loss function L is always the squared difference between predicted and actual output.

Neural Network regression The linear and logistic regression neuron representation can be extended to neural networks as in Figure 2.10:

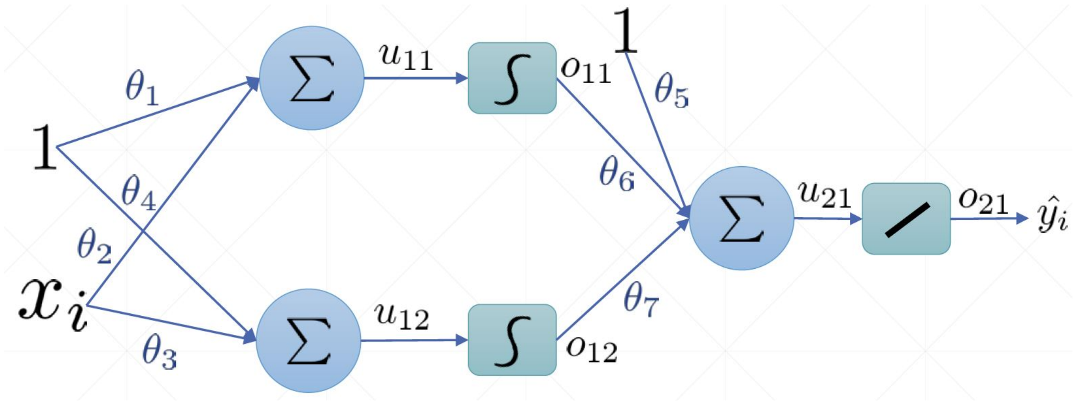


Figure 2.10: A Neural networks on 1 dimensional input data set.

From Figure 2.10, first the linear regression model which is a weighted combination of inputs and weights are used. Followed by a sigmoid activation function ($\sigma(x) = \frac{1}{1+e^{-x}}$), which says if the neuron has to be activated or not (1 if activated). The θ_1 and θ_2 are weights or parameters here. Ultimately, the final layer has linear activation because we are interested in using neural networks for regression.

In Fig. 2.10, the output $\hat{y}_i$ is given by,

$$\hat{y}_i = \theta_5 + \theta_6\sigma(\theta_1 + \theta_2 x_i) + \theta_7\sigma(\theta_4 + \theta_3 x_i) \tag{2.8}$$

Where,

- $\theta's$ are the parameters.
- x_i is the input of the i^{th} sample
- Activation function (sigma represented in green circular rectangle) is the sigmoid function ($\sigma(x) = \frac{1}{1+e^{-x}}$)
- u_{11}, u_{12} are the outputs of the neurons before activation
- o_{11}, o_{12} are the outputs of the neurons after activation in first layer
- o_{21} is the output of the neuron after activation in the second layer
- Since linear activation is used in final layer, $\hat{y}_i = o_{21} = u_{21}$

Now our goal is to learn the parameters θ of the neural network. If g is the loss function, the parameters are computed using the gradient descent update given as,

$$\theta^{t+1} = \theta^t - \alpha \times \frac{\partial g(\theta)}{\partial \theta} \tag{2.9}$$

The gradient of loss, g with respect to the parameter is given as,

$$\frac{\partial g(\theta)}{\partial \theta_j} = -2(y_i - \hat{y}_i(x_i, \theta))\frac{\partial \hat{y}_i(x_i, \theta)}{\partial \theta_j} \tag{2.10}$$

The value of $\hat{y}_i(x_i, \theta)$ can be computed while doing a forward pass (discussed below) and the derivative of $\hat{y}_i$ with respect to the parameter is computed during the back propagation step discussed below.

2.3.1 Forward Pass

In forward pass, each element of θ has a known value. Then from Fig. 2.10, the values of u, o and $\hat{y}_i$ are given as,

- $u_{11} = \theta_1 \times 1 + \theta_2 \times x_i$
- $u_{12} = \theta_4 \times 1 + \theta_3 \times x_i$
- $o_{11} = \frac{1}{1+e^{-u_{11}}}$
- $o_{12} = \frac{1}{1+e^{-u_{12}}}$
- $u_{21} = \theta_5 \times 1 + \theta_6 \times o_{11} + \theta_7 \times o_{12}$
- $\hat{y}_i = o_{21} = u_{21}$

2.3.2 Backpropagation

The next step is to compute the partial derivatives of $\hat{y}_i$ with respect to the parameter θ. Each $\hat{y}_i$ can be represented as a function of previous layer's weights as,

$$\hat{y}_i = \theta_5 \times 1 + \theta_6 \times o_{11} + \theta_7 \times o_{12} \tag{2.11}$$

From the above equation the gradient of $\hat{y}_i$ with respect to $\theta_5, \theta_6, \theta_7$ is given as,

- $\frac{\partial \hat{y}_i}{\partial \theta_5} = 1$
- $\frac{\partial \hat{y}_i}{\partial \theta_6} = o_{11}$
- $\frac{\partial \hat{y}_i}{\partial \theta_7} = o_{12}$

Similarly, $u_{12} = \theta_2 \times x_i + \theta_3 \times 1$, now the derivative of $\hat{y}_i$ with respect to θ_3 is given in equation (2.12). It is essential to note that θ_3 only influences $\hat{y}_i$ through u_{12}.

$$\frac{\partial \hat{y}_i}{\partial \theta_3} = \frac{\partial \hat{y}_i}{\partial o_{12}} \frac{\partial o_{12}}{\partial u_{12}} \frac{\partial u_{12}}{\partial \theta_3} \tag{2.12}$$

The $o's$ in the equation 2.12 are received during the forward pass. Similarly the gradient with respect to $\theta_1, \theta_2, \theta_4$ are computed.

Once the $\hat{y}_i$ and the derivatives of $\hat{y}_i$ with respect to the θ are computed, the gradient descent as given in 2.9 is used to update the parameters θ and the forward pass and update step continues until the optimum value of parameter (or minimal loss) is attained.

Neural Network implementation via graph object Since the computation above involves backpropagating errors to update the parameters, the entire operation is constructed in the form of graph to organize the computation better. The operations can be thought as a linear list of operations. To make it more precise the function that maps from input to output can be imagined within a directed acyclic graph (DAG) of operations, wherein the data and parameters vector flow along the edges and the nodes represent differentiable transformations. The implementation of backpropogation in recent deep learning frameworks like tensorflow and pytorch uses a graph object that records the connectivity operations (gates and layers) along with list of operations. Each graph and node object has two functions as discussed earlier: `forward()`, for doing the forward pass and `backward()` for performing the gradient descent update using backpropogation. The `forward()` goes over all the nodes in a topological order and takes computation flow in forward direction to predict the output using the known weights. `backward()` goes over nodes in reverse direction and does the update step. Each node computes its output with some function during its `forward()` operation, and in `backward()` operation, the nodes are given the gradient of loss function with respect to edges (weights) attached to the nodes. The derivatives are done using finite difference method. The backward process lists the gradient on output and multiplying it by its own local gradient and this gradient then flows to its child nodes. The child nodes then perform the same operation recursively.

Let us illustrate by describing the implementation of a tanh layer. The forward function is given as $y = \tanh x$. The backward function will have $\frac{\partial g}{\partial y}$ (Jacobian product) given by the *Graph* object, where g is the loss at the end of the graph. The backward function chains the Jacobian of this layer onto the running product. Here, the product is the matrix product $\frac{\partial g}{\partial x} = \frac{\partial y}{\partial x}\frac{\partial g}{\partial y}$. Now, since the Jacobian $\frac{\partial y}{\partial x}$ of this layer only has elements along the diagonal, we can implement this efficiently by rewriting the matrix multiplication as $\frac{\partial g}{\partial x} = \mathbf{eye}(\frac{\partial y}{\partial x}) \odot \frac{\partial g}{\partial y}$, where $\odot$ is an element wise multiplication and $\mathbf{eye}(\frac{\partial y}{\partial x})$ is the diagonal of the matrix as a vector (for tanh this would be the vector $(1 - y \odot y)$ since $\frac{d}{dx}\text{tanh}x = (1 - (\text{tanh}x)^2)$. The returned $\frac{\partial g}{\partial x}$ via the Graph object from this layer will be passed on to the node that computed x. The recursion continues till it reaches the inputs of the graph.

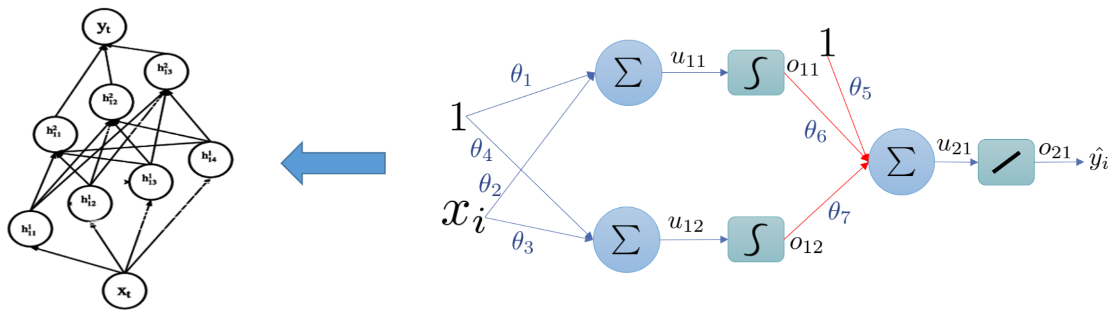


Figure 2.11: A different representation of neural network as used in literature.

Figure 2.11 shows a different representation of neural network as used in literature. It will be useful to understand recurrent neural networks in the later part of the chapter. Note: the neural network in the right hand side and the left hand side are not exactly equal, the motivation of this figure is to show how neural network could be expressed just using circles.

2.4 Neural Networks

We have seen how we can use neural networks to perform a regression task. We defined a differentiable function that transform the inputs x to predicted outputs $\hat{y}$. Then we optimized the model with respect to loss and parameters using gradient descent methods. Now we will see more details about the objective and extend to recurrent neural networks and Long Short Term Memory (LSTM).

2.4.1 Simple Neural Networks

Neural networks involve series of repeated matrix multiplications and element-wise non-linearities. Some of the commonly used non-linearities are,

- Sigmoid, $\sigma(x) = 1/(1 + e^{-x})$
- ReLU, $\sigma(x) = max\{0, x\}$
- tanh, $\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

For a vector $\vec{x} = (x_1, ..., x_N)$, we write $\sigma(\vec{x}) = (\sigma(x_1), ..., \sigma(x_N)$ as the natural element-wise extension. For instance, a 2-layer neural network would be implemented as $f(x) = W_2\sigma(W_1x)$, where σ is an element-wise non-linearity (e.g. sigmoid) and W_1, W_2 are matrices. 3-layer will take the following form $f(x) = W_3\sigma(W_2\sigma(W_1x))$, etc. If the non-linearity is not used in between the layers then the entire neural network becomes a simple linear model, because the weights in the each layer get multiplied by the weights in the next, forming a series of matrix products, which is just a single weighted combination of input, which is just linear. The last layer of the neural network either has a sigmoid or a non-linear layer to squash the input to 0 and 1 for classification tasks or identity function in the case of regression problems.

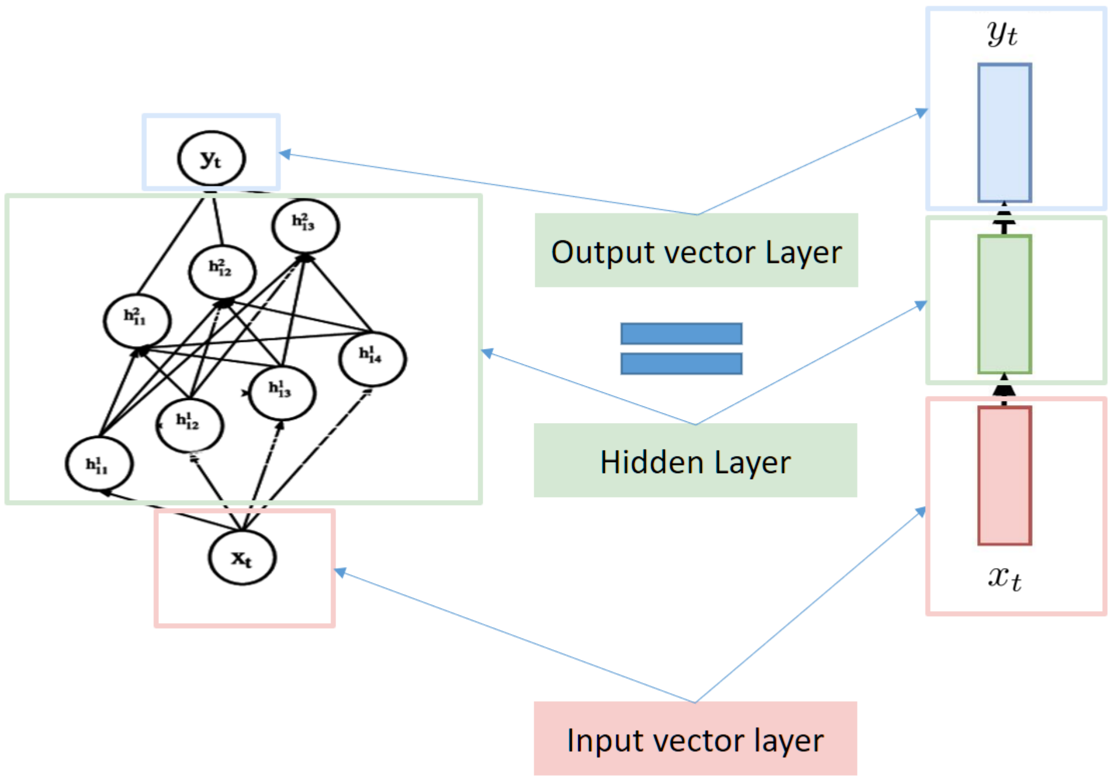


Figure 2.12: Compact way of representing neural networks with just input layer, hidden layer and output layer.

Figure 2.12 will be useful in Chapter 3 in constructing complex recurrent neural networks which will be discussed later in this chapter.

Biological inspiration A crude model of biological neurons is the motivation for neural network. Each row of the parameter or weight matrices W corresponds to one neuron and its synaptic connection strengths to its input. The positive weights relate to excitation, whereas the negative weights relates to inhibition. On the other hand, the weight zero represents no dependence. The weighted sum of the inputs and the weights reaches the neurons at the cell body. The sigmoid activation function draws motivation from the fact that, since it squashes the output to [0, 1] it can be interpreted as indicating if a neuron has fired or not.

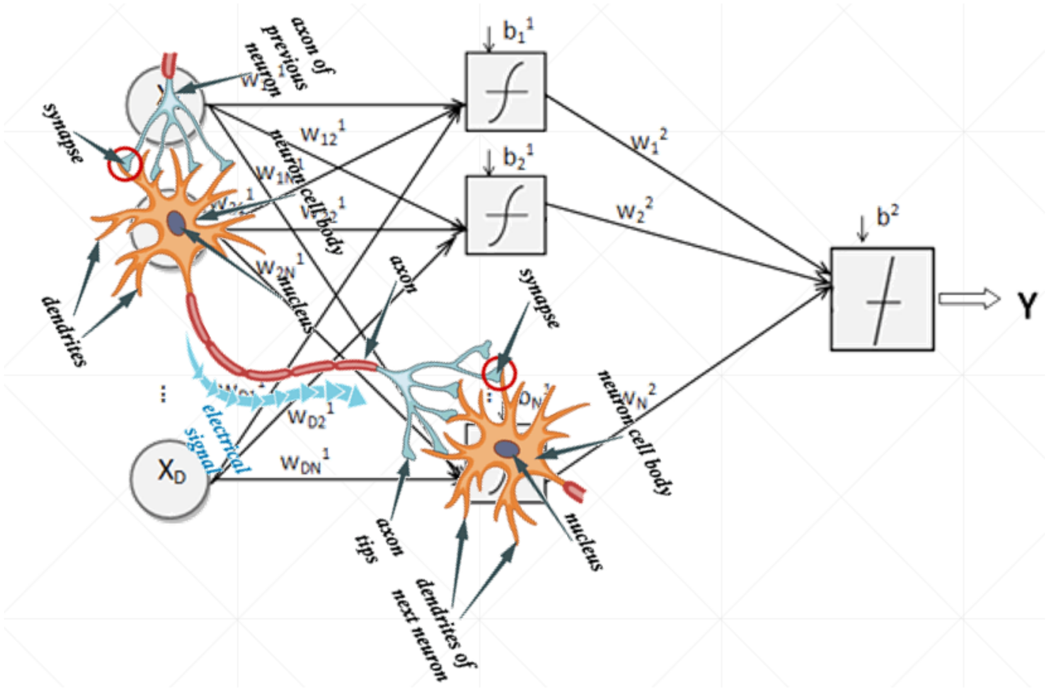


Figure 2.13: The real neural network in the brain is merged and connected with the artificial neural network in this figure. [49], [50]

From Figure 2.13, the Artificial Neural Networks got inspiration from the neural network in the brain. The connection between the two is given by the edges and represents weights or parameter of the neural network model. The weights in Artificial Neural Network helps in transferring signal (0 means does not transfer signal) to other neurons.

2.4.2 Recurrent Neural Networks

Recurrent neural networks are machine learning models that inherit neural network architecture to handle input and output sequences. Weather prediction requires time series of temperature, pressure data in the form of sequences to be processed as inputs so as to predict the future temperature. A recurrent neural network (RNN) uses a recurrence formula of the form $h_t = f_\theta(h_{t-1}, x_t)$, where f is a function that we describe in more detail below and the same parameters θ are used at every time step. The function f_θ is a connectivity pattern that can process a sequence of vector $x_1, x_2, ..., x_T$, with arbitrary length. The hidden vector h_t can be thought of as a running summary of all vectors $x_{t'}$ for $t' < t$. The recurrence formula updates the summary based on the latest input. For initialization, we can either treat h_0 as vector parameters and learn the starting hidden state or simply set $h_0 = \vec{0}$. There are many forms of recurrence and the exact mathematical form of the recurrence $(h_{t-1}, x_t) \mapsto h_t$ varies from neural network architecture to architecture. We will discuss about this later in this section.

The simplest Recurrent Neural Network uses a recurrence of the form,

$$h_t = tanh\big(W \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix} \big), \tag{2.13}$$

where the *tanh* function is applied element-by-element transformation as outlined in section 2.4.1.

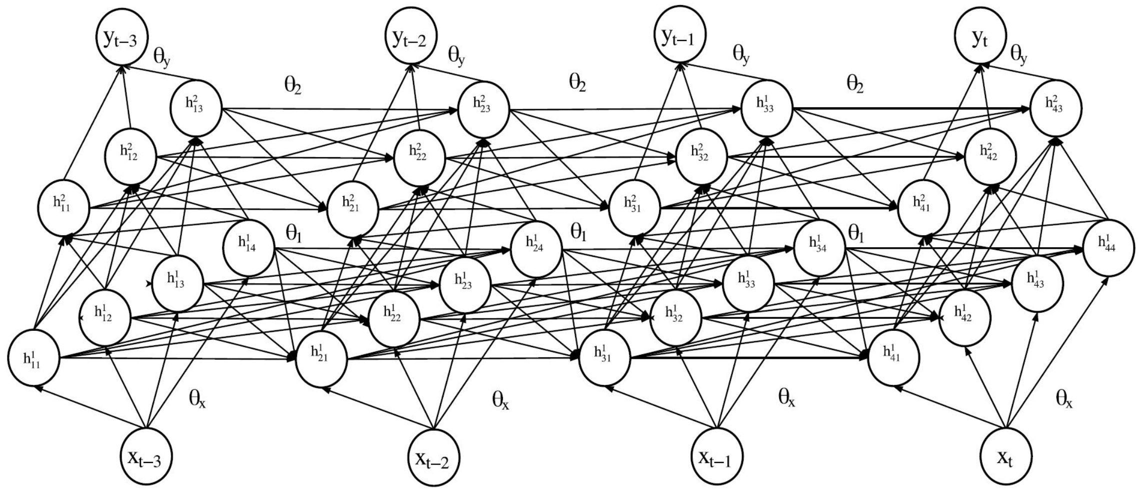


Figure 2.14: A recurrent neural network

From Figure 2.14, the neural network can be thought of as rolling feed forward neural network in time. From the above figure you can see the that the θ_x is preserved as the same parameter in each and every time step. The same works for other parameters θ's along the time step. The reason why it stays constant is that, while training because of limitation in memory the network can be trained only with constant time steps, because of the involvement of many gradients along the entire time step. While testing since the same parameter remains in each time step the neural network can be rolled to any steps in the future for for future prediction.

The current input and the previous hidden vector are concatenated and transformed linearly by the matrix product with W. This is equivalent to $h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1})$, where the two matrices W_{xh}, W_{hh} concatenated horizontally are equivalent to the matrix W above. These equations do not consider the bias vector that should be included. Instead of the tanh non linearity, a ReLU or sigmoid non linearity can also be used. If the input vectors x_t have dimension D and the hidden states have dimension H then W is a matrix of size $[H \times (D + H)]$. Interpreting the equation, the new hidden states at each time step are a linear combination of elements of x_t, h_{t-1}, squashed by non-linearity. The simple RNN has a simple neat form, but the additive and simple multiplicative interactions result in weak coupling [22, 23] especially between the inputs and the hidden states. The functional form of the simple RNN leads to undesirable dynamics during backpropagation [24]. The gradients either explode (if the gradient value is greater than 1) or vanish (if the gradient value is less than 1) especially when expanded over long time periods. This specific exploding gradient problem can be overcoming with gradient clipping [25], i.e., clipping the gradient or restarting the gradient especially if it reaches a threshold. On the other hand, the vanishing gradient problem cannot be addressed by gradient clipping.

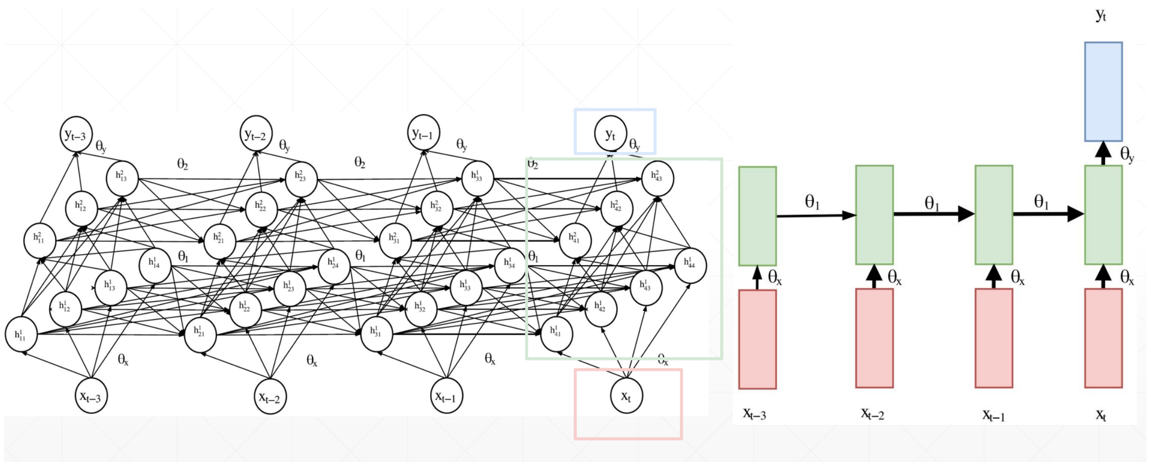


Figure 2.15: Block representation of RNN. RED: input layer, GREEN: hidden layer, BLUE: output layer.

2.4.3 Long Short-Term Memory

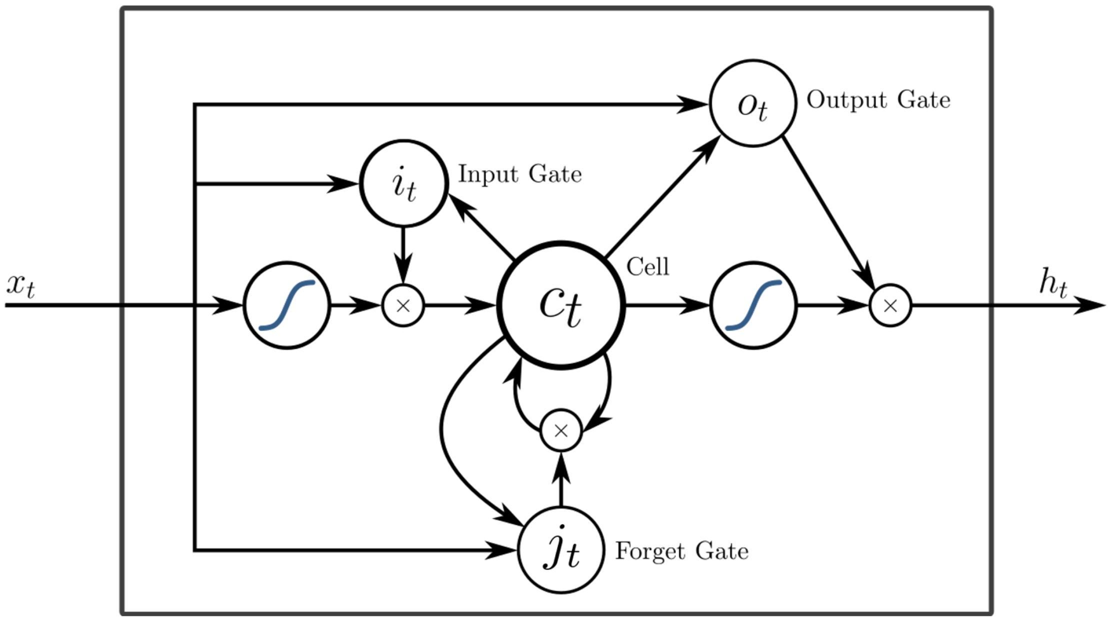


Figure 2.16: LSTM internal structure [30]

The motivation for Long Short Term Memory [26] (LSTM) is to address some of the limitations of RNN, especially during training, i.e., vanishing and exploding gradient problem. Its recurrence formula allows the inputs x_t and h_{t-1} to interact in a more computationally complex manner. The interaction includes multiplicative interactions and additive interactions over time steps that effectively propagate gradients backwards in time [26]. LSTMs maintain a memory vector c_t in addition to a hidden state vector h_t. The LSTM can choose to read from, write to, or reset the cell using explicit gating mechanisms at each time step . The LSTM recurrence update is as follows:

$$\begin{bmatrix} i \\ j \\ o \\ g \end{bmatrix} = \begin{bmatrix} sigm \\ sigm \\ sigm \\ tanh \end{bmatrix} W \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix} \tag{2.14}$$

$$c_t = f \odot c_{t-1} + i \odot g \tag{2.15}$$

$$h_t = o \odot tanh(c_t) \tag{2.16}$$

If the input dimensionality is D and the hidden state has H units then the matrix W has dimensions $[4H \times (D + H)]$. Here, the sigmoid function sigm and tanh are applied element-wise. The $i \in \mathcal{R}^H$ is a binary gate that controls whether each memory cell is updated. The $j \in \mathcal{R}^H$ is a forget gate that controls whether it is reset to zero and the $o \in \mathcal{R}^H$ is an output gate controls whether its local state is revealed in the hidden state vector. To keep the model differentiable and hence allowed to range smoothly between zero and one, the activations of these gates are based on the sigmoid function . The vector $g \in \mathcal{R}^H$ ranges between -1 and 1 and is used to additively modify the memory contents. This unique additive interaction operation makes LSTM stand out of simple RNN. During backpropagation this additive operation equally distributes gradients. This allows gradients on c to flow backwards in time for long time periods without vanishing or exploding.

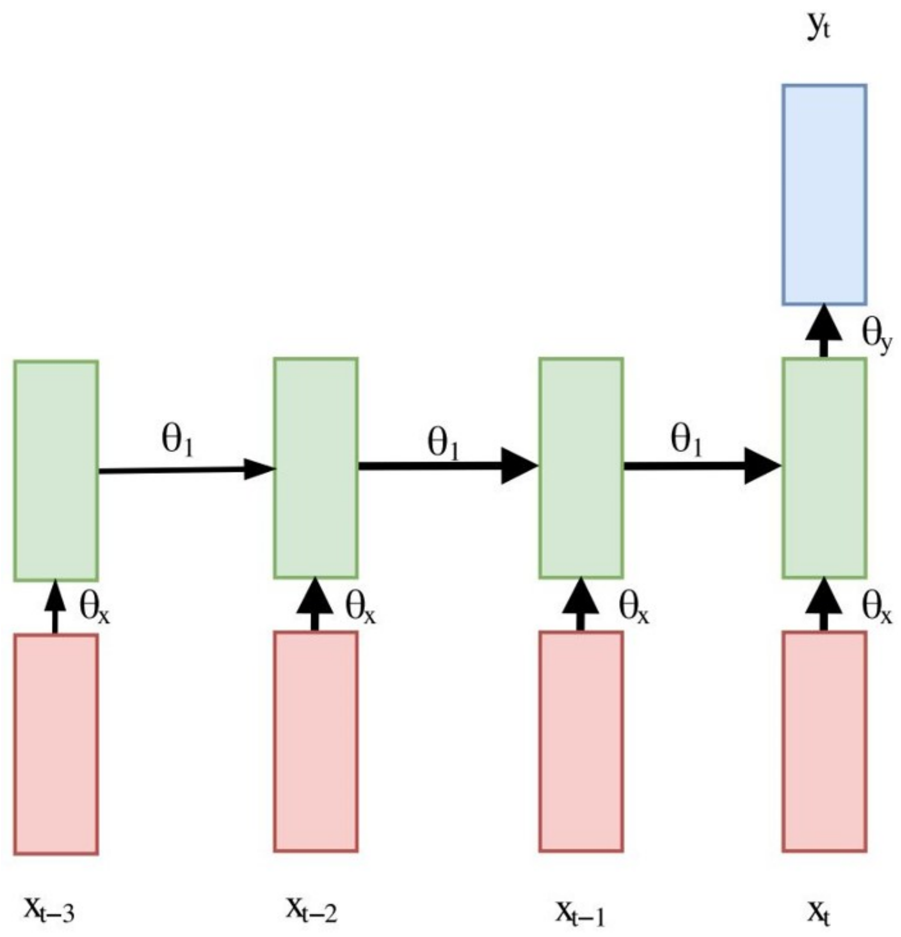


Figure 2.17: Sample RNN architecture for time series prediction.

Time series prediction with RNN Figure 2.17 shows a sample RNN architecture for time series based prediction. Consider a data set where we have time series data of daily temperatures over the last couple of years and our goal is to predict temperature at the next day given the previous 4 days of temperature. Now, the input can be fed as series of windows of size 4 and the output to train the network is the temperature at the next day after the day 4 i.e., 4 days of temperature as input from x_{t-3} to x_t

and the next day temperature as output, y_t. Here, the loss function will be $L(\theta) = \sum_{i=1}^{n}(y_t^i - x_{t+1}^i)^2$, where y_t^i is the predicted output from RNN and x_{t+1}^i is the labeled output in the training data.

The first part of our work discussed in Chapter III involves learning of control behaviour using supervised learning. Having seen machine learning, specifically supervised learning, and the various models and optimization techniques involved in supervised learning, we now turn to Model Predictive Control, the knowledge of which is essential for chapter III.

2.5 Model Predictive Control

Model (Based) Predictive Control (MBPC or MPC) originated in the late 1970's and has been constantly developed since then, from simple prediction strategies to the incorporation of economics into MPC constraints. The term Model Predictive Control explicitly uses a model of the process to obtain the control signal by minimizing an objective function. It does not designate a specific control strategy or a control algorithm but a very broad range of control methods. MPC design and formulation lead to non-linear controllers which present adequate degrees of freedom and have practically the same structure as that of MPC. In general the predictive control family involves the following key ideas:

- Uses a model (empirical or first principle) to predict the process output for some number of future time steps (or horizon).

- Calculates a sequence of control actions (equal to control horizon) by minimizing an objective function.

- Using the prediction step uses the future error, to come up with the first control action to the plant, then re-calculating as above. This strategy is called receding strategy.

The different MPC algorithms that have been developed vary by the model that is used to represent the process, the noise model, and the cost function to be minimized. MPC has got its application in diverse set of industries ranging from chemical process industries to clinical industries.

2.5.1 MPC Strategy

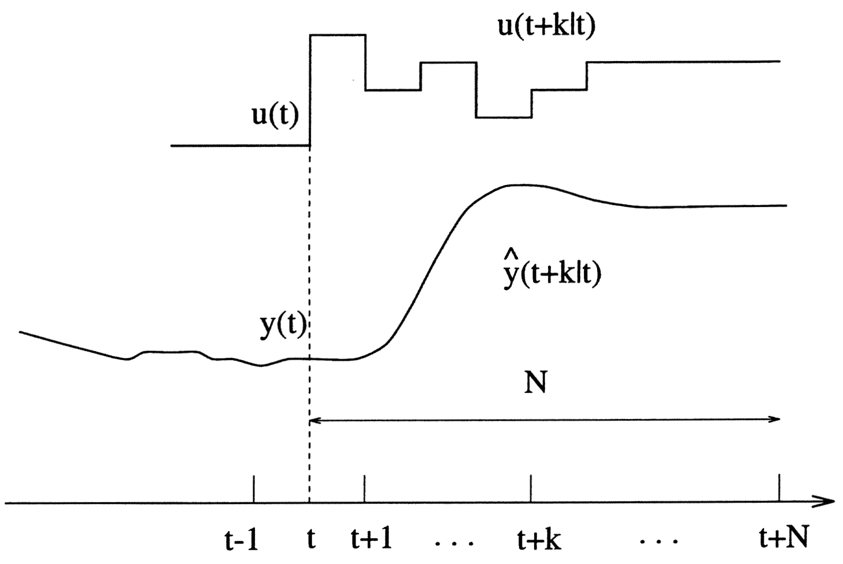


Figure 2.18: MPC strategy. [52]

Now let us see the strategies of MPC that makes this controller a well suited one for control.

1. The prediction horizon which is the future outputs for a predefined horizon N are predicted and computed at each instant t using the defined process model. The predicted outputs $y(t+k|t)$ for $k = 1, ..., N$ depend on the known values up to instant t and on the future control inputs $u(t+k|t)$, $k = 0, ..., N-1$, that are to be sent to the plant and to be calculated.

2. The goal is to keep the process output close to the reference set point or trajectory $w(t+k)$, hence future control input is calculated by optimizing an objective function. The objective function is usually a quadratic function - commonly the squared error in difference between the predicted output and the reference set point. The control effort and other constraints (economical or operational constraints) are included in the objective function. Even for simple quadratic objective, the presence of constraints means an iterative optimization method has to be used. Performance is tuned in the objective function, by choosing weights for different process outputs especially if the system is Multi Input Multi Output system.

3. Once the control input $u(t|t)$ is sent to the process, the next control signals calculated from minimizing the objective function are discarded. The reason is at the next instant $y(t+1)$ is already known and step 1 can be repeated with the new value that is returned by optimization objective function. Then all the sequences are brought up to date, and $u(t+1|t+1)$ is calculated using the receding horizon concept, discussed earlier.

2.5.2 MPC Components

To set up an MPC, we must stipulate the horizon, an explicit model and optimization method.

- Horizon
- Explicit use of model
- Optimization

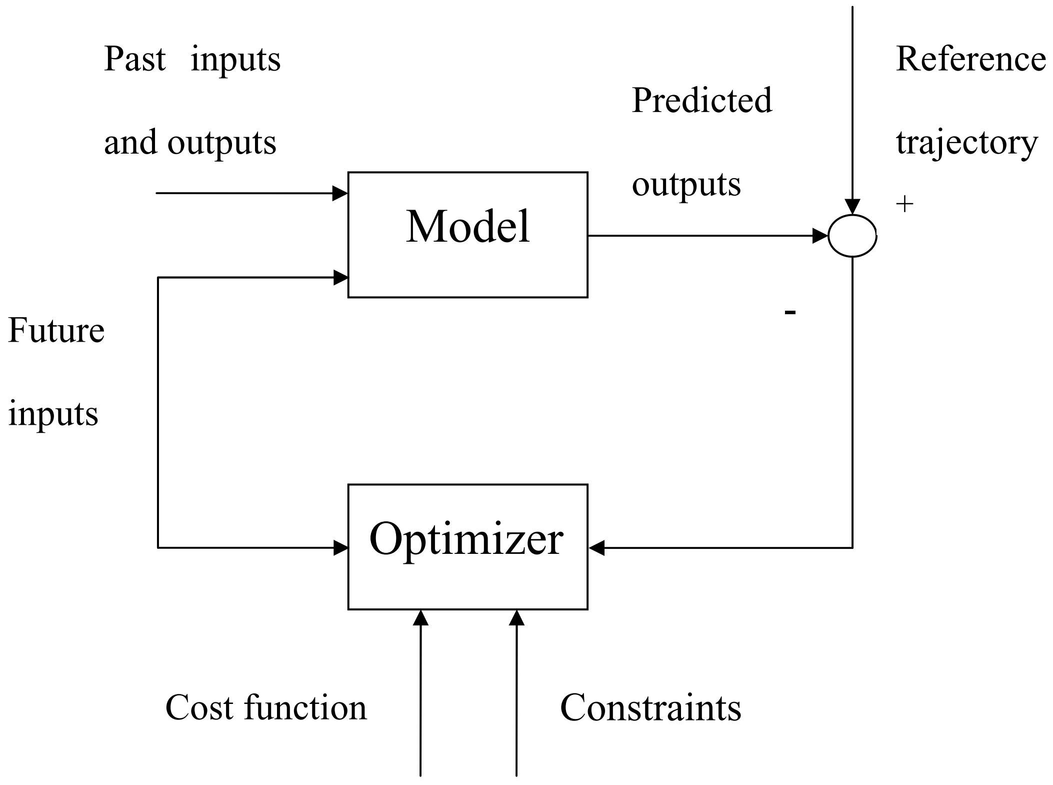


Figure 2.19: Basic structure of MPC. [52]

Standard Model

The standard model we use in MPC is a state space model given by,

$$X_{k+1} = AX_k + BU_k \tag{2.17}$$

$$Y_k = CX_k + DU_k \tag{2.18}$$

where A, B, C, D are constant matrices whose dimensions are compatible with the dimensions of the state X, contol U, and output Y.

Horizon

MPC uses prediction horizon (number of time steps to look forward) to predict forward and control horizon (number of times steps that we have control on) to take control actions. The prediction horizon and control horizon can be visualized using the figure 2.20,

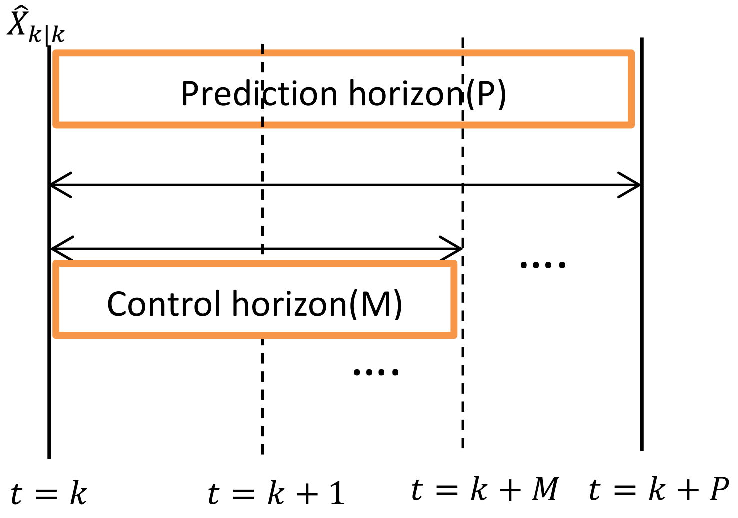


Figure 2.20: Prediction and Control Horizon

Hence if we denote P as prediction horizon and M as control horizon, the state space model can be rolled forward with respect time steps as,

$$\begin{aligned}
X_{k+1} &= AX_k + BU_k \\
X_{k+1}^{\hat{}}|k &= A\hat{X}_k + BU_k \\
X_{k+2}^{\hat{}}|k &= AX_{k+1|k}^{\hat{}} + BU_{k+1} \\
X_{k+M+1}^{\hat{}}|k &= AX_{k+M|k}^{\hat{}} + BU_{k+M} \\
X_{k+M+2}^{\hat{}}|k &= AX_{k+M+1|k}^{\hat{}} + BU_{k+M}
\end{aligned}$$

After the control horizon, the input does not change so the BU_{k+M} is common. The decision variables in the optimization problem are the control values at all time instants up to control horizon.

Optimization Objective MPC

We define,

$$\hat{Y}_{k:k+P} = \begin{bmatrix} \hat{Y}_{k+1} \\ \hat{Y}_{k+2} \\ . \\ . \\ \hat{Y}_{k+P} \end{bmatrix}, \hat{Y}^{ref}_{k:k+P} = \begin{bmatrix} \hat{Y}^{ref}_{k+1} \\ \hat{Y}^{ref}_{k+2} \\ . \\ . \\ \hat{Y}^{ref}_{k+P} \end{bmatrix}, U_{k:k+M} = \begin{bmatrix} U_k \\ U_{k+1} \\ . \\ . \\ U_{k+M} \end{bmatrix} \tag{2.19}$$

,

where $\hat{Y}$ is the predicted output, $\hat{Y}^{ref}$ is the reference set point and U is the control input.

Objective function of MPC can be written as,

$$(\hat{Y}_{k:k+P} - \hat{Y}^{ref}_{k:k+P})^T H (\hat{Y}_{k:k+P} - \hat{Y}^{ref}_{k:k+P}) + U^T_{k:k+M} Q U_{k:k+M}$$

Where H and Q is a weighting matrix. . Either U_k or $\Delta U_k = U_{k+1} - U_k$ can be used as decision variables.

Constraints To protect the physical actuators and to avoid excessive fluctuations, input constraints are imposed. The constraints are given as,

$$U^{min} \leq U \leq U^{max} \tag{2.20}$$

$$\Delta U^{min} \leq \Delta U \leq \Delta U^{max} \tag{2.21}$$

We also impose constraints on the output for stability.

$$Y^{min} \leq Y \leq Y^{max}$$

Coincidence Points If we want the output to reach a certain value at a certain time, we can pose this requirement as an equality constraint. The definition of the coincident point is $Y^{ref}_{k+i} = Y^{SP}$, where Y^{SP} is the set point.

Tuning parameters Available tuning parameters are M, P, H and W. P is chosen based on the steady state of the system under consideration. H and W are to be chosen based on the system knowledge and we can tune H, W and M. Infinite control horizon problems are stable [56].

Implementation of MPC The way we implement MPC is at $t = k$, from optimization problem we get $U_k, U_{k+1}, ..U_{k_M}$. We take U_k and implement that control move. At the next time step, we obtain additional information Y_{k+1}, so the $U_k' s$ are recalculated and the solution of an optimization problem will provide $U_{k+1}, ..U_{k+M}$. We will implement U_{k+1} control move alone and discard the rest. This procedure is repeated till we reach control horizon. After the control horizon, we don't perform any control moves. To be implemented online in a real plant, when the sampling time is very small, the optimization calculations should be fast enough to give control moves at the rates compatible with the sampling rates. The optimization problems required to implement MPC are Quadratic programming problems, for which fast solution algorithms are well developed.

Handling of Model Mismatch If one can assume that the difference in the predicted and the measured Y values is due to the model mismatch d_k alone, the offset in the estimated states can be reduced by including d_k with the predicted values of Y_k at each time instant, with the assumption that $d_{k+i} = d_k$, k to $k + M$, and when we move to next step, use d_{k+1}, newly calculated and follow the assumption for the time steps ahead up to M. This correction is equivalent to using the integrating effect, in a PI controller and this will eliminate the offset.

The second part of our work discussed in Chapter IV involves learning of control behavior using reinforcement learning. Earlier we saw about machine learning and MPC. Now we will discuss reinforcement learning, which will be applied to process control in chapter IV, where we develop a learning based controller.

2.6 Reinforcement Learning

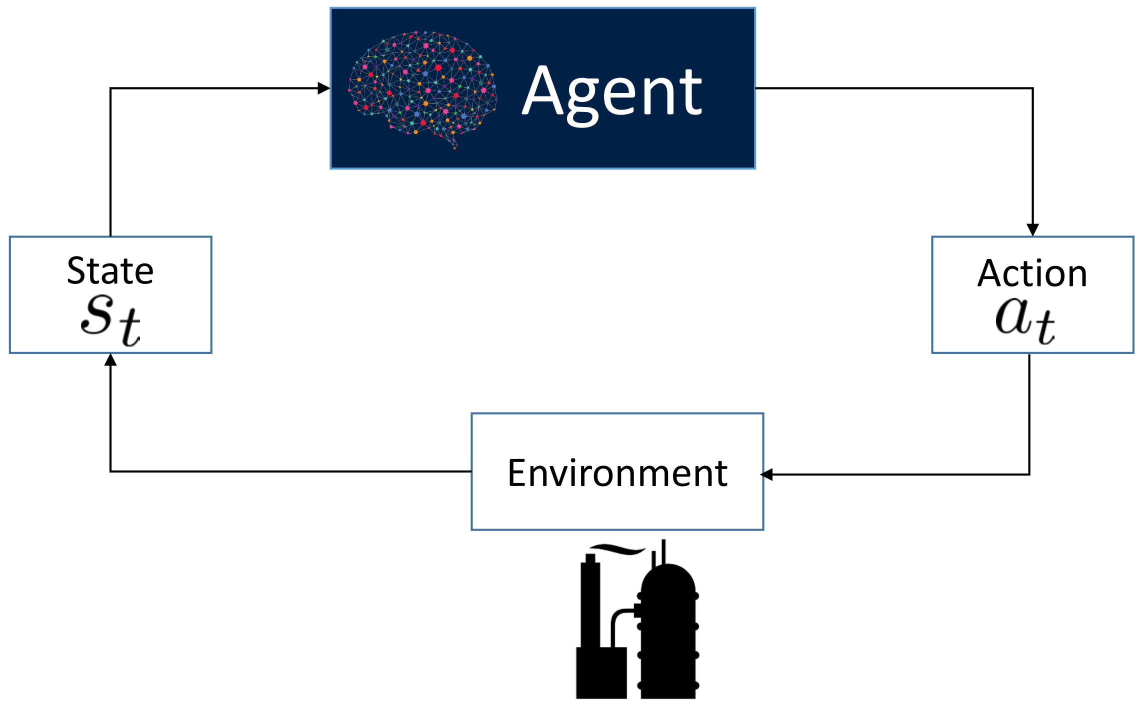


Figure 2.21: Reinforcement learning flow diagram

2.6.1 Introduction

Reinforcement learning is entirely driven by a reward hypothesis, for which details will be discussed later. It is a sequential decision making problem in which an agent takes an action at time step t and learns as it performs that action at each and every time step, with the goal of maximizing the cumulative reward. The decision maker is the agent and the place the agent interacts or acts is the environment. The agent receives observations (or states) from the environment and takes an action according to a desired (or to be learnt) policy at each time step. Once the action is performed, the environment then provides a reward signal (scalar) that indicates how well the agent has performed. The general goal of reinforcement learning is to maximize the agent's future reward, given all past experience of the agent in the environment. The reinforcement learning problem is usually modelled as a Markov Decision Process (MDP) that comprises a state space and action space, reward function and an initial state distribution that satisfies the Markov property that $P(s_{t+1}, r_{t+1}|s_1, a_1, r_1, ...s_t, a_t, r_t) = P(s_{t+1}, r_{t+1}|s_t, a_t)$. where s_t, a_t and r_t are states, action and reward at time t respectively and P is a joint distribution conditioned on state, action and reward until time t.

2.6.2 Markov Decision Processes

Markov Decision Processes (MDPs) are used to better explain or to model the environment where the agent interacts. In our work and in this dissertation we consider fully observable MDPs. The MDP is defined as a tuple $(S, D, A, P_{sa}, \gamma, R)$ consisting of :

- S: Set of possible states of the world
- D: An initial state distribution
- A: Set of possible actions that the agent can perform
- P_{sa}: The state transition probabilities (which depend on the action)
- γ: A constant in (0, 1) that is called the discount factor, (used to give less effect on future reward and more weight to the current reward)

- R: A scalar-valued reward function defined on $S \times A \rightarrow R$, which indicates how well the agent is doing at each time step.

In an MDP, there is an initial state s_0 that is drawn from the initial state distribution. The agent interacts in the environment through action, and at each time step, resulting in the next successor state s_{t+1}. At each step by taking actions, the system visits a series of states $s_0, s_1, s_2, ...$ in the environment such that the sum of discounted rewards as the agent visits is given by:

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + ... \tag{2.22}$$

The goal of the reinforcement learning agent is to take actions $a_0, a_1, ...$ such that the expected value of the rewards given in equation (2.22) is maximized. The agent's goal is to learn a good policy (which is a mapping from states to actions) such that during each time step for a given state, the agent can take a stochastic or deterministic action $\pi(s)$ that will result in maximum expected sum of rewards, compared to other possible sequences of actions:

$$E_\pi[R(s_0, \pi(s_0)) + \gamma R(s_1, \pi(s_1)) + \gamma^2 R(s_2, \pi(s_2)) + ...] \tag{2.23}$$

2.6.3 Learning Formulation

Here we consider only model-free reinforcement learning methods. The model-free reinforcement learning framework aims at directly learning the system from past experience without any prior knowledge of the environment dynamics. A policy could be a function or a discrete mapping from states to actions and is used to select actions in a MDP. We consider both stochastic

$a \sim \pi_\theta(a|s)$ and deterministic $a = \pi_\theta(s)$ policies. Usually function based policies are used because if the state and action space is large then the discrete mapping (in the form of a table) is not efficient to capture all the states and actions. Hence, we consider function based parametric policies, in which we choose an action a (whether stochastically or deterministically) according to a parameter vector $\theta \in \mathbb{R}^n$. Given parametric policy, the agent can interact with the MDP to give a series of states, actions and rewards $h_{1:T} = s_1, a_1, r_1, ...s_T, a_T, r_T$. We use the return, r_t^γ which is the total discounted reward from time step t onwards, $r_t^\gamma = \sum_{k=t}^{\infty} \gamma^{k-t} r(s_k, a_k)$ where $0 < \gamma < 1$. The agent's objective function is given by $J(\pi) = \mathbb{E}[r_1^\gamma|\pi]$. In most process control applications, the state and action spaces are continuous hence we use neural networks to approximate the policy.

2.6.4 Expected Return

As discussed earlier, the goal of reinforcement learning is to maximize the expected return of a policy π_θ with respect to the expected cumulative return.

$$J(\theta) = Z_\gamma \mathbb{E} \sum_{k=0}^{H} \gamma^k r_k \tag{2.24}$$

where $\gamma \in (0, 1)$ is the discount factor, H is the planning horizon, and Z_γ is a normalization factor. To achieve this, we want to find a policy that can help us achieve the maximum reward. Here, we consider actor-critic based policy gradient algorithms. The policy gradient algorithms can handle continuous state and action spaces. They adjust the parameters θ of the policy in the direction of the performance gradient $\nabla_\theta J_{\pi_\theta}$ maximizing the objective function $J(\pi_\theta)$ which is the value function (value function will be discussed later).

Reinforcement Learning problems can be formulated in two ways, Value based and policy based reinforcement learning. We now briefly describe both options.

2.6.5 Value-Based Reinforcement Learning

Reinforcement learning uses the value functions which is used to evaluate goodness/ badness of states. For an agent following a policy $\pi(s)$, the value function $V^{\pi}(s)$ is given by:

$$V^{\pi}(s) = E_{\pi}[R_t | s_t = s] \tag{2.25}$$

To incorporate action in the value function, action value function is used. It is used to evaluate goodness/ badness of states along with the actions. The action value function $Q^{\pi}(s, a)$ when the agent takes action a for state s, and following a policy π, is given by:

$$Q^{\pi}(s, a) = E_{\pi}[R_t | s_t = s, a_t = a] \tag{2.26}$$

where E_{π} denotes the expectation over agent's experiences that are collected by agent following the policy π. As stated earlier, we consider model-free reinforcement learning such that the value or action-value function (which is a neural network) is updated using a sample backup as the agent interacts. This means, that at each time step of the agent's interaction, an action is sampled from the agent's current policy. Then the successor state is sampled from the environment and the corresponding rewards are computed. The action value function is updated as the agent gets reward by interacting with the environment.

2.6.6 Policy based Reinforcement Learning

The value based approach above can handle discrete states and actions. For continuous states and actions an optimization based approach is required to learn the continuous policy function. Policy gradient algorithms rely upon optimizing parameterized policies with respect to the future cumulative reward. Policy gradient directly optimize a parametrized (parametrized with neural network) policy by gradient ascent such that the optimal policy will maximize the agent's cumulative reward at each and every time step. For Value-based methods local optima convergence is not guaranteed, hence policy gradient methods are useful because they are guaranteed to converge in the worst case at least to a locally optimal policy. Also the policy gradient algorithms can handle high dimensional continuous states and actions. The disadvantage of policy gradient algorithms is that it is difficult to attain a globally optimal policy.

In this work, we consider deterministic policy gradient algorithm for learning the control behaviour of the plant. Hence, let us investigate the details of the deterministic policy gradient approach.

2.6.7 Policy Gradient Theorem

The policy gradient theorem [58] is the basis for a series of deterministic policy gradient algorithms. We first consider stochastic policy gradient theorem, then gradually migrate from probabilities and stochasticity to a more deterministic framework. The stochastic policy gradient theorem, says that for stochastic policies, one has

$$\nabla_\theta J(\theta) = E_{s\sim\rho^\pi, a\sim\pi_\theta[\nabla_\theta log\pi_\theta(s,a)Q^{\pi_\theta}(s,a)]} \tag{2.27}$$

The deterministic policy gradient theorem says that for deterministic policy is given as:

$$\nabla_\theta J(\theta) = E_{s\sim\rho^\pi}[\nabla_{\theta\mu_\theta(s)}\nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}] \tag{2.28}$$

2.6.8 Gradient Ascent in Policy Space

Since we are interested in maximizing the expected return by choice of policy instead of gradient descent which searches for minimum, we have to search for the maximum using gradient ascent. The gradient step to increase the expected return is given as,

$$\theta_{k+1} = \theta_k + \alpha_k \nabla_\theta J(\theta)|_{\theta=\theta_k} \tag{2.29}$$

Here θ_0 is the initial policy parameter and θ_k denotes the policy parameter after k updates. α_k denotes the learning rate which is related to step-size.

2.6.9 Stochastic Policy Gradient

The policy gradient algorithm searches for a locally optimal policy (locally optimum) by ascending the gradient of $J(\theta)$ of the policy, such that, $\delta\theta = \alpha\nabla_\theta J(\theta)$, and α is the learning rate for gradient ascent. If we consider a Gaussian policy whose mean is a linear combination of state features $\mu(s) = \phi(s)^T\theta$.

In the equation of the stochastic policy gradient theorem expressed in equation 2.27, the equation for the $\nabla_\theta log\pi_\theta(s,a)$, that is called the score function, is given by:

$$\nabla_\theta log\pi_\theta(s,a) = \frac{(a-\mu(s))\phi(s)}{\sigma^2} \tag{2.30}$$

From equation 2.27, the $Q^{\pi_\theta}(s,a)$ is approximated using a linear function approximator (linear regression or linear regression with basis functions), where the critic estimates $Q^w(s,a)$, and in the stochastic policy gradient, the function approximator is compatible such that

$$Q^w(s,a) = \nabla_\theta log\pi_\theta(a|s)^T w \tag{2.31}$$

where the w parameters are chosen to minimize the mean squared error between the true and the approximated action-value function.

$$\nabla_\theta J(\theta) = E_{s\sim\rho^\pi, a\sim\pi_\theta}[\nabla_\theta log\pi_\theta(s,a)Q^w(s,a)] \tag{2.32}$$

If we use the off-policy stochastic actor-critic along with the stochastic gradients setting. we get the following expression for the stochastic off-policy policy gradient:

$$\nabla_\theta J(\theta)_\beta(\pi_\theta) = E_{s\sim\rho^\beta, a\sim\beta}[\frac{\pi_\theta(a|s)}{\beta_\theta(a|s)}\nabla_\theta log\pi_\theta(a|s)Q^\pi(s,a)] \tag{2.33}$$

where $\beta(a|s)$ is the behaviour off-policy (that is used during learning) which is different from the parameterized stochastic policy, $\beta(a|s) = \pi_\theta(a|s)$ that is being learnt.

2.6.10 Deterministic Policy Gradient

The stochastic gradient framework can be extended to deterministic policy gradients [12] by removing the stochasticity involved, similar to the reduction from the stochastic policy gradient theorem to the deterministic policy gradient theorem. In deterministic policy gradients having continuous actions, the goal is to move the policy in the direction of gradient of Q rather than taking the policy to the global maximum. Following that in the deterministic policy gradients, the parameters are updated as follows:

$$\theta^{k+1} = \theta^{k} + \alpha E_{s \sim \rho^{\mu^k}}[\nabla_{\theta} Q^{\mu_k}(s, \mu_{\theta}(s))] \tag{2.34}$$

The function approximator $Q^w(s, a)$ is consistent with the deterministic policy $\mu_\theta(s)$ if :

$$\nabla_a Q^w(s, a)|_{a=\mu_\theta(s)} = \nabla^T_{\theta\mu_\theta(s)} w \tag{2.35}$$

where $\nabla_{\theta\mu_\theta(s)}$ is the gradient of the actor function approximator. Using the function approximator, we get the expression for the deterministic policy gradient theorem as shown in the equation below:

$$\nabla_\theta J(\theta) = E_{s\sim\rho^\beta}[\nabla_{\theta\mu_\theta(s)} \nabla_a Q^w(s, a)|_{a=\mu_\theta(s)}] \tag{2.36}$$

We consider the off-policy deterministic actor-critic where we learn a deterministic target policy $\mu_\theta(s)$ from trajectories to estimate the gradient using a stochastic behaviour policy $\pi(s, a)$ that is considered only during training.

$$\nabla_\theta J_\beta(\mu_\theta) = E_{s\sim\rho^\beta}[\nabla_{\theta\mu_\theta(s)} \nabla_a Q^w(s, a)|_{a=\mu_\theta(s)}] \tag{2.37}$$

2.7 Function Approximation

The action-value function has to be learnt in reinforcement learning setup. If the number of states and actions are large, it is impossible to store them all in a table. Hence, function approximators are used. Since, deep learning found a great success in the recent years (after breakthroughs in image classification, speech recognition, machine translation, etc.), deep neural networks are used as a function approximators to learn the action-value function, i.e., to approximate Q in the policy gradient theorem. Policy gradient methods use function approximators in which the policy is represented by a function approximator and is updated with respect to the policy parameters during the learning process by following a stochastic policy.

In this dissertation, we have only used neural networks as function approximators. Both the stochastic and the deterministic policies can be approximated with different neural network approximators, with different parameters w for each network. The main reason for using function approximators is to learn the action value function that cannot be stored in a gigantic table especially when the number of states and actions are large. To learn action value function using function approximators, policy evaluation methods like temporal difference learning are used.

2.7.1 Temporal Difference Learning

We use Temporal Difference (TD) Learning to find the action value function. TD learning learns the action value function using reward, current and successive states and actions. Temporal Difference methods have been widely used policy evaluation reinforcement learning based algorithm. TD learning applies when the transition probability P and the reward function r^π are not known and we simulate the RL agent with a policy π.

The off-policy TD algorithm known as Q-learning was one of the famous model-free reinforcement learning based algorithms [59]. Here the learnt action value function is a direct estimate of the optimal action-value function Q^*. For learning the action value function, again a loss has to be defined and we will have squared error loss with reward and from action value at next time step. This squared loss will be useful for learning the parameters of action value function neural network. It will be discussed in detail in the next section.

2.7.2 Least-Squares Temporal Difference Learning

The least squares temporal difference algorithm (LSTD) [51] takes advantage of sample complexity and learns the action value function quickly. As discussed earlier, in order to avoid storing many state action pair in a table, for large state and action spaces, for both continuous and discrete MDPs, we approximate the Q function with a parametric function approximator (neural network) as follows:

$$\hat{Q^{\pi}} = \sum_{i=1}^{k} \pi_i(s,a)w_i = \phi(s,a)^T w \tag{2.38}$$

where w is the set of weights or parameters, ϕ is a $(|S||A| \times k)$ matrix where row i is the vector $\phi_i(s,a)^T$, and we find the set of weights w that can yield a fixed point in the value function space such that the approximate Q action-value function becomes:

$$\hat{Q^{\pi}} = \phi w^{\pi} \tag{2.39}$$

Assuming that the columns of ϕ are independent, we require that $Aw^{\pi} = b$ where b is $b = \phi^T R$ and A is such that

$$A = \phi^T(\phi - \gamma P^{\pi}\phi) \tag{2.40}$$

We can find the function approximator parameters w from:

$$w^{\pi} = A^{-1}b \tag{2.41}$$

2.8 Actor-Critic Algorithms

In policy gradient actor-critic [31], the actor adjusts the policy parameters θ by gradient ascent to maximize the action value function, for both stochastic and deterministic gradients. The critic learns the w parameters of the LSTD function approximator to estimate the action-value function $Q^w(s,a) \approxeq Q^{\pi}(s,a)$. The estimated $Q^w(s,a)$ that we substitute instead of the true $Q^{\pi}(s,a)$ may introduce bias in our estimates if we do not ensure that the function approximator is compatible for both stochastic and deterministic gradients.

The function approximator for stochastic policy gradient is $Q^w(s,a) = \nabla_{\theta} log \pi_{\theta}(a|s)^T w$ and the function approximator for deterministic policy gradient is $Q^w(s,a) = w^T a^T \nabla_{\theta_{\mu_\theta(s)}}$, where $\nabla_{\theta_{\mu_\theta(s)}}$ is the gradient of the actor. Our approach in this work is off policy actor critic. To estimate the policy gradient using an off-policy setting we use a different behaviour policy to sample the trajectories. We use the off-policy actor critic algorithm to sample the trajectories for each gradient update using a stochastic policy. With the same trajectories, the critic also simultaneously learns the action-value function off-policy by temporal difference learning.

2.9 Exploration in Deterministic Policy Gradient Algorithms

Since we are learning a deterministic policy while learning the reinforcement learning agent our agent does not explore well. Hence, stochastic off-policy in a deterministic setting will make sure that there is enough exploration in the state and action space. The advantage of off-policy exploration is that the agent will learn a deterministic policy in deterministic policy gradients but will choose its actions according to a stochastic behaviour off-policy. The exploration rate should be set carefully, domain knowledge is required to set the exploration rate. Also, manual tuning of exploration (σ parameters) can also help the agent learn to reach a better policy, as it can explore more. In our work, we consider obtaining trajectory of states and actions using stochastic off-policy (which makes sure of enough exploration) and thereby learn a deterministic policy.

2.10 Improving Convergence of Deterministic Policy Gradients

We used the following optimization based techniques to improve the convergence of deterministic policy gradient algorithms.

2.10.1 Momentum-based Gradient Ascent

Classic Momentum

The classic momentum technique was already seen in the first order based derivative optimization section in section 2.3. Classical momentum (CM) for optimizing gradient descent update, is a technique for accelerating gradient ascent. It can accumulate a velocity vector in directions of consistent improvement in the objective function. For our objective function $J(\theta)$, the classical momentum approach is given by:

$$v_{t+1} = \mu v_t + \epsilon \nabla_\theta J(\theta_t) \tag{2.42}$$

$$\theta_{t+1} = \theta_t + v_{t+1} \tag{2.43}$$

where, θ is our vector of policy parameters, $\epsilon > 0$ is the learning rate, and we use a decaying learning rate of $\epsilon = \frac{a}{t+b}$, where a and b are the parameters that we run a grid search over to find optimal values, t is the number of learning trials on each task, and μ is the momentum parameter $\mu \in [0, 1]$.

2.10.2 Nesterov Accelerated Gradient

We also used Nesterov Accelerated Gradient (NAG) for optimizing gradient descent update in our policy gradient algorithms. It is a first order gradient based method that has a better convergence rate guarantee. The gradient update of NAG is given as:

$$v_{t+1} = \mu v_t + \epsilon \nabla_\theta J(\theta_t + \mu v_t) \tag{2.44}$$

We used both classic momentum and nesterov accelerated gradient to speed up the learning.

2.11 Summary

Let us summarize the work flow involved in training a machine learning based model.

Data preparation The first task is to collect training data to be able to start training the model. The input data is represented as x and output data is represented as y in this dissertation. For validating our model we split the data into three sets. 80% data as training data, 10% as validation data and the remaining 10% as test data. We use gradient descent based optimization technique to learn the parameters from the training set. For verifying the hyper parameters, the validation data set will be used. Finally to evaluate the performance of the model, the test data set will be used.

Data preprocessing Data preprocessing helps in rapidly finding the optimal parameters. In this dissertation, the mean shift was done to have zero mean and the individual samples of the features were divided by the variance, to have variance of 1. Also, chapter 3 involves training for recurrent neural networks, which involved processing of data as time steps in the form of window slices.

Architecture Design Coming up with an effective neural architecture involves understanding the underlying structure with supporting theories.

In process control based applications, data are stored with respect to time. A common neural network model to learn time series data is the recurrent neural network. Hence, in Chapter 3 we have proposed a recurrent neural network based architecture to learn optimal control behaviour of MPC. The number of layers and the number of hidden units can be chosen manually to minimize the validation error.

Optimization Throughout the dissertation the optimization technique used for performing gradient descent update is the Adam optimizer [21] with learning rate of 10^{-3}. Along with Nesterov (NAG) and Momentum (CM) based technique to perform gradient ascent while computing action value function. We also made sure that the learning rate decays over time, we achieved this by multiplying the learning rate by 0.1 after every 200 iterations. To make sure our optimization algorithm actually searches for minima, we run the algorithm so as to over fit the data and get 0 training error. In this way we confirm that bug did not exist in the code. We also store weight files once every few iterations, in this way you can decide to choose the weights that had lower validation error.

Hyper parameter Optimization There is no deterministic way of coming up with better hyper parameters. The best way to choose hyper parameters is to randomly choose different hyper parameters and then pick the hyper parameter that has minimum training and validation error.

Evaluation Once the model is trained on the training data set, it has to be tested on the test data set, to check for test error of the trained machine learning model. Once we identify the best hyper parameter using hyper parameter optimization, we can use the hyper parameter to learn the entire sample (with training, validation and test data combined.) The machine learning model is then ready for evaluation and testing.

The reminder of this dissertation is organized around the two projects that leverage the application of deep learning in process control to solve some of its important challenges.

Chapter 3

Offline Policy Learning - A deep learning architecture for predictive control

Referred Conference Publication:
The contributions of this chapter have been submitted in:
Steven Spielberg Pon Kumar, Bhushan Gopaluni, Philip Loewen,*A deep learning architecture for predictive control*, Advanced Control of Chemical Processes, 2018 [submitted].
Conference Presentation:
The contributions of this chapter have also been presented in:
Steven Spielberg Pon Kumar, Bhushan Gopaluni, Philip Loewen,*A deep learning architecture for predictive control*, Canadian Chemical Engineering Conference, Alberta, 2017.

In this chapter we develop approaches for learning the complex behaviour of Model Predictive Control (MPC). The learning is done offline. In the next chapter we will see how to learn the complex control policies just by interacting with the environment. Model Predictive Control (MPC) is a popular control strategy that computes control action by solving an optimization objective. However, application of MPC can be computationally demanding and sometimes requires estimation of the hidden states of the system, which itself can be rather challenging. In this work, we propose a novel Deep Neural Network (NN) architecture by combining standard Long Short Term Memory (LSTM) architecture with that of NN to learn control policies from a model predictive controller. The proposed architecture, referred to as LSTM supported NN (LSTMSNN), simultaneously accounts for past and present behaviour of the system to learn the complex control policies of MPC. The proposed neural network architecture is trained using an MPC and it learns and operates extremely fast a control policy that maps system output directly to control action without having to estimate the states.We evaluated our trained model on varying target outputs, various initial conditions and compared it with other trained models which only use NN or LSTM.

3.1 Introduction

Controlling complex dynamical systems in the presence of uncertainty is a challenging problem. These challenges arise due to non-linearities, disturbances, multivariate interactions and model uncertainties. A control method well-suited to handle these challenges is MPC. In MPC, the control actions are computed by solving an optimization problem that minimizes a cost function while accounting for system dynamics (using a prediction model) and satisfying input-output constraints. MPC is robust to modeling errors [29] and has the ability to use high-level optimization objectives [32]. However, solving the optimization problem in real time is computationally demanding and often takes lot of time for complex systems. Moreover, MPC sometimes requires the estimation of hidden system states which can be challenging in complex stochastic non-linear systems and in systems which are not observable. Also, standard MPC algorithms are not designed to automatically adapt the controller to model plant mismatch. Several algorithms exist to speed up the MPC optimization through linearization [33] of the non-linear system and through approximation of the complex system by a simpler system [33],[34],[35]. However, these algorithms do not account for the complete non-linear dynamics of the system. In this chapter, we propose using the deep neural network function approximator to represent the complex system and the corresponding MPC control policy. Once the control policies of MPC are learned by the proposed deep neural network, no optimization or estimation is required. The deep neural network is computationally less demanding and runs extremely fast at the time of implementation.

We propose a novel neural network architecture using standard LSTM supported by NN models (LSTMSNN). The output of LSTMSNN is a weighted combination of the outputs from LSTM and NN. This novel neural network architecture is developed in such a way that its output depends on past control actions, the current system output and the target output. The LSTM part of LSTMSNN architecture uses past control actions as input to capture the temporal dependency between control actions. The NN part of LSTMSNN architecture uses the current system output and the target output as inputs to predict the control action. Our simulations show that the proposed deep neural network architecture is able to learn the complex nonlinear control policies arising out of the prediction and optimization steps of MPC.

We use a supervised learning approach to train LSTMSNN. First, we use MPC to generate optimal control actions and system output for a given target trajectory. Then, we use these data to train LSTMSNN and learn the MPC control policies. Once the control policies are learned, the LSTMSNN model can completely replace the MPC. This approach has the advantage of not having to solve any online optimization problems. In this chapter, we restrict ourselves to developing the LSTMSNN model.

Our main contribution is the combined LSTM and NN architecture that can keep track of past control actions and present system information to take near optimal control actions. Since LSTMSNN uses deep neural networks accounting for past and present information, we can train complex, high-dimensional states (system with large number of states) in stochastic non-linear systems using this approach. One of the unique features of our method is that the network is trained using an MPC. The trained LSTMSNN Model is computationally less expensive than MPC because it does not involve an optimization step and does not require estimation of hidden states. In implementation, we use a Graphical Processing Unit (GPU) to parallelize the computation and the prediction of optimal control actions is done rather rapidly.

3.2 Related Work

Model predictive control (MPC) is an industrially successful algorithm for control of dynamic systems [37],[38] but it suffers from high computational complexity when controlling large dimensional complex non-linear systems - (1) the optimization step that determines the optimal control action can take a very long time, especially for higher dimensional non-linear systems; (2) the estimation of states can be burdensome. To overcome this, various functional approximation techniques were developed for use with MPC [39],[40]. In addition, model free methods such as reinforcement learning techniques [42] were also developed. NN were used to approximate the MPC prediction step [34] and the optimization cost function. NN were also used to approximate the nonlinear system dynamics [35] and then MPC was performed on the neural network model. However, NN generated system predictions tend to be oscillatory. Other approaches include [36] using a two tiered recurrent neural network for solving the optimization problem based on linear and quadratic programming formulations. However, these approaches involve estimation of hidden states at each instant. LSTMs are effective at capturing long term temporal patterns [30] and are used in a number of applications like speech recognition, smart text completion, time series prediction, etc. Our approach differs from others in that LSTMSNN is based on the past MPC control actions and the current system output. Moreover, our approach does not involve the burden of estimating the hidden states that characterize system dynamics.

3.3 Model

In this work, we propose a novel neural network architecture called LSTM-SNN. We use LSTMSNN (weighted linear combination of LSTM and NN) to learn the complex behaviour of MPC. Block diagrams for the training and testing phases are shown in Figure 3.1 and Figure 3.2. In the training phase shown in Figure 3.1, the LSTMSNN neural network model learns the MPC controlling the system through the data from set point, MPC optimal control action and plant output. Once the model has learnt, it can be used to generate optimal control actions required to control the plant as given in Figure 3.2.

The different neural network models used for training MPC behaviour are discussed below.

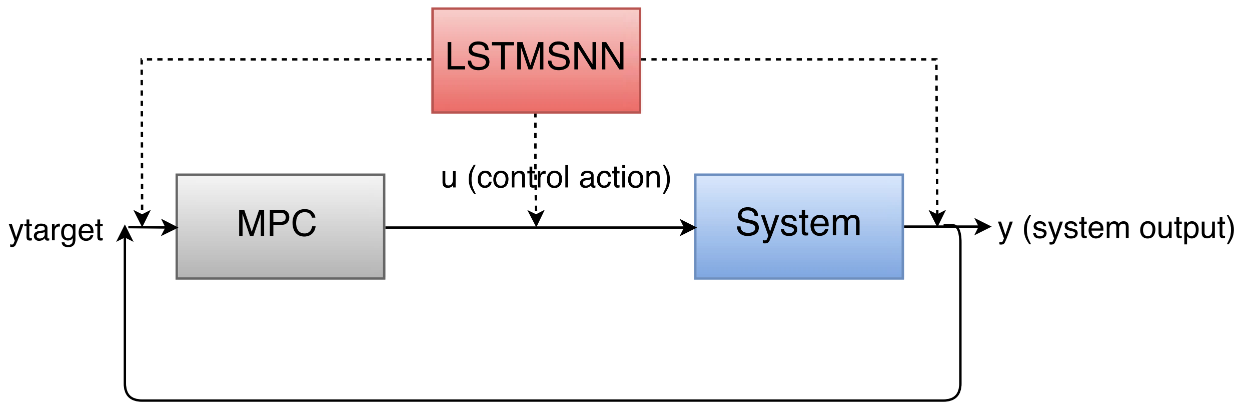


Figure 3.1: Computation flow of the training process

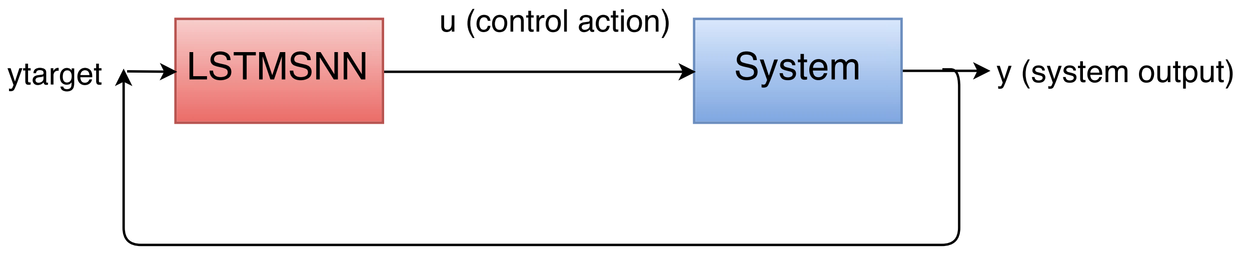


Figure 3.2: Computation flow using the trained model to control the system.

3.3.1 Long Short Term Memory (LSTM)

The LSTM with a sequence length of 5 is trained with data generated from MPC. There is a helpful overview of LSTM in chapter 2 section 2.4.3. The inputs to the model are past MPC control actions, plant output and target. The output is the control action to be taken at the next time step. Fig. 3.3 illustrates the LSTM model. In Fig. 3.3, the ℓ denotes the final feature values to be considered in the LSTM sequence. For instance, $u[k-\ell]$ denotes the final past control action in the LSTM sequence. The details of the model is given in table 3.2

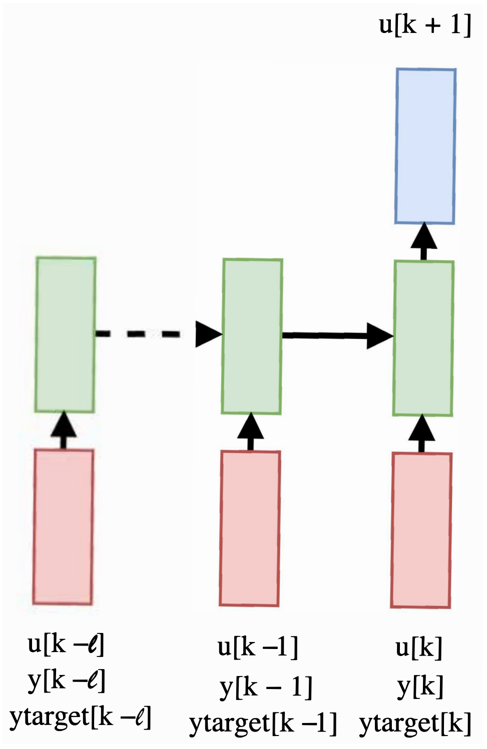


Figure 3.3: Architecture of LSTM-only model.

3.3.2 Neural Network (NN)

The inputs to our NN are the previous system output and target output. The output of NN is the control action at the next time step. The considered NN model has 3 layers. Fig. 3.4 illustrates the NN model. The details of the NN model architecture is given in table 3.2

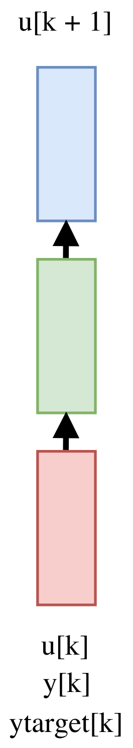


Figure 3.4: Architecture of NN-only model

3.3.3 LSTMSNN

This is the new proposed architecture combining the LSTM (sequence length of 5) and NN (3 layers) part. The motivation for this design is MPC control actions depend on both current system output and past input trajectories.

Hence, the LSTMSNN output is a weighted combination of the LSTM (which takes past input into account) and the NN (which uses the current plant output and required set point) to learn the optimal control action given past input trajectories, current plant output and required set point. The best configuration of LSTMSNN resulted after tuning by simulation is shown in Fig. 3.5. The details of the LSTMSNN model is given in table 3.2

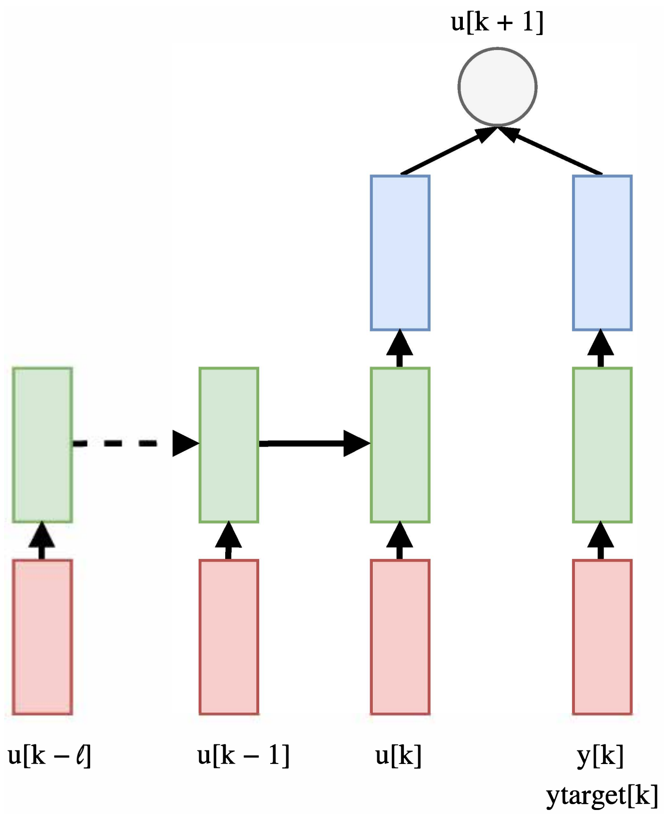


Figure 3.5: LSTMSNN model

3.4 Experimental setup

3.4.1 Physical system

The physical system we chose to study is the manufacture of paper in a paper machine. The target output is the desired moisture content of the paper sheet. The control action is the steam flow rate. The system output is the current moisture content. The difference equation of the system is given by,

$$y[k] = 0.6 \times y[k-1] + 0.05 \times u[k-4] \tag{3.1}$$

The corresponding transfer function of the system is,

$$G(z) = \frac{0.05z^{-4}}{1 - 0.6z^{-1}} \tag{3.2}$$

The time step used for simulation is 1 second.

3.4.2 Implementation

The neural networks are trained and tested using the deep learning framework Tensorflow [41]. In Tensorflow, the networks are constructed in the form of a graph and the gradients of the network required during training are computed using automatic differentiation. Training and testing were done on GPU (NVIDIA Geforce 960M) using CUDA [6]. We ran the MPC and generated artificial training data in Matlab.

3.4.3 Data collection

We required three variables for training: the target output, system output, and control action. All the data used for training was artificially generated in

Matlab. We chose the target outputs and then used MPC to find the optimal control actions and resulting system outputs. There are three different sets of data used in this study. Each set of data had 100,000 training points.

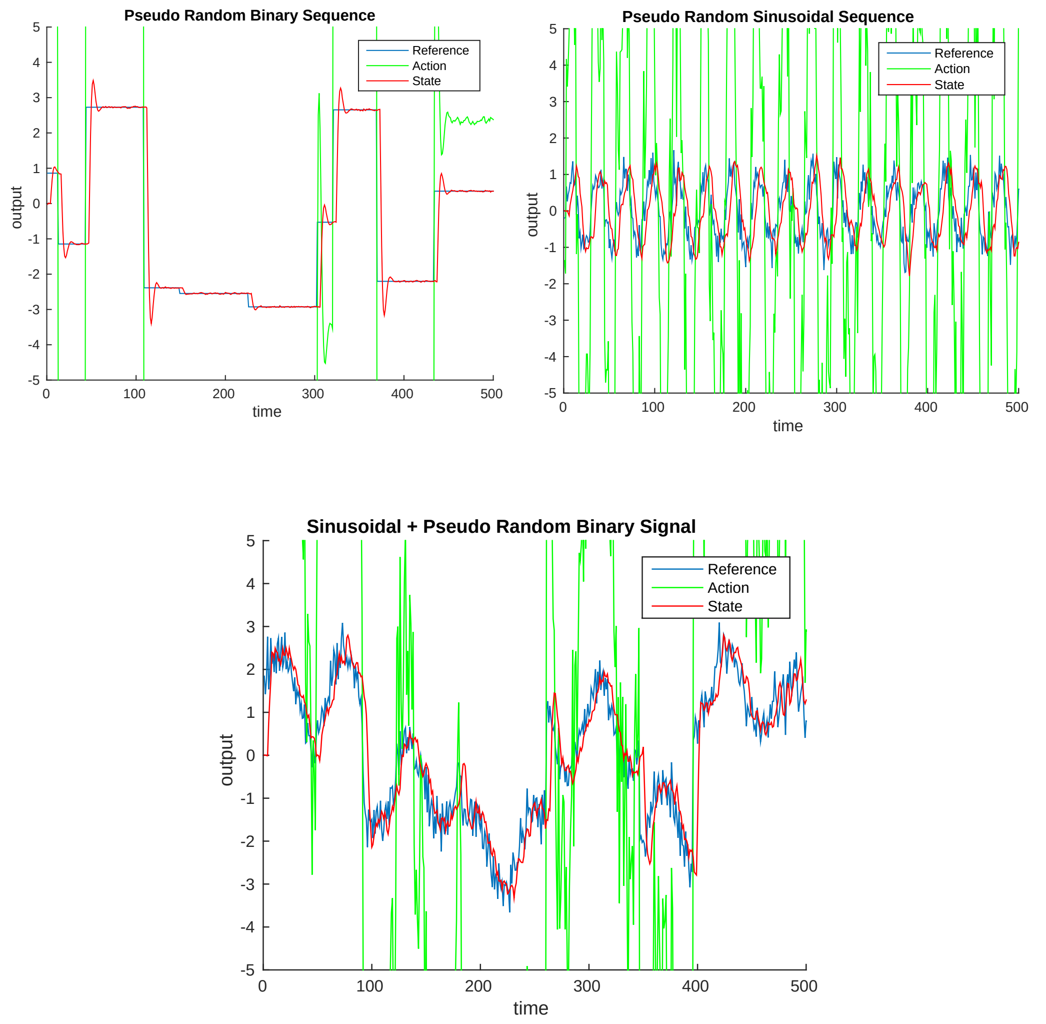


Figure 3.7: Excerpts of length 500 from three types of training data. Blue is the target output, red the system output and green the control action. The binary, sinusoidal, and combined sequences are shown on the top left, top right, and bottom respectively

Table 3.1: MPC details

Variable	**Value**
Platform	Matlab (Model Predictive Control Tool Box)
Prediction horizon	20
Control horizon	10
Sampling time	1 s

The first set is a pseudorandom binary sequence (RandomJump). The target output randomly jumps to a value in the range $(-3, -0.2) \cup (0.2, 3)$. The target output then remains constant for 10 to 100 time steps before jumping again. The second set is a pseudorandom sinusoidal sequence. For each 1000 time steps, a sine function with period $(10, 1000)$ time steps was chosen. Gaussian noise was added to the data with signal-to-noise ratio of 10. The last set of data is the combination of the first two sets, a pseudo-random binary-sinusoidal sequence (SineAndJump). At each time step, the target outputs from the two data sets were added. The motivation for using pseudorandom binary data to train is that a step function is the building block for more complex functions. Any function can be approximated by a series of binary steps. Furthermore, physical systems are often operated with a step function reference. The motivation to train on sinusoidal data is that our model should learn periodic behaviour.

The details of MPC setup is as follows:

3.5 Simulation Results

We conducted experiments to show the effects of data set on LSTMSNN's performance, the effectiveness of LSTMSNN relative to other models, and the robustness of LSTMSNN under various initial conditions and target outputs.

We trained LSTMSNN by using RMSprop. We also used the two parts of LSTMSNN, LSTM and NN, as comparison models in our experiments. They are denoted as LSTM-only and NN-only. Details of the model architectures are in Table 3.2. Note that LSTMSNN uses a NN with many layers because we found that a simpler structure does not perform well during test time. We think this is because our training data sets are large, which limits a simpler structure from getting the most out of training data. However, decreasing the size of the training data decreases LSTMSNN's performances during test time, especially under varying target outputs. So there is a trade off between the complexity of the architecture and the size of the training data.

Table 3.2: Details of comparison models

Model	Optimizer	Number of layers	Sequence Length	Learning rate
LSTMSNN	RMSprop	2 layers of LSTM and 4 layers of NN	5	0.001
LSTM-only	RMSprop	2	5	0.001
NN-only	Stochastic Gradient Descent	3	N/A	0.001

Our first set of experiments shows the effect of training data on LSTMSNN's performance. We test the performance of LSTMSNN by specifying an initial condition. LSTMSNN will generate a control action in each time step and get a system output. Previous control action, system output and a constant target output will be the input to LSTMSNN in the next time step. We run this process for 1000 time steps. We use mean square error (MSE) and offset error (OE) to measure the performance. MSE is the average of squared errors between the system model's output and the target output during the process. OE is the difference between system model's output and the target output after the system output converges. We will compare the performances of LSTMSNN by training on RandomJump and SineAndJump, as shown in Table 3.3.

Table 3.3: Performance comparison by testing different data set

Data Set	Target Output	MSE	OE
RandomJump	2	0.02	0.01
SineAndJump	2	0.06	0.01
RandomJump	5	0.176	0.18
SineAndJump	5	0.078	0.01
RandomJump	10	1.28	1.8
SineAndJump	10	0.01	0.01

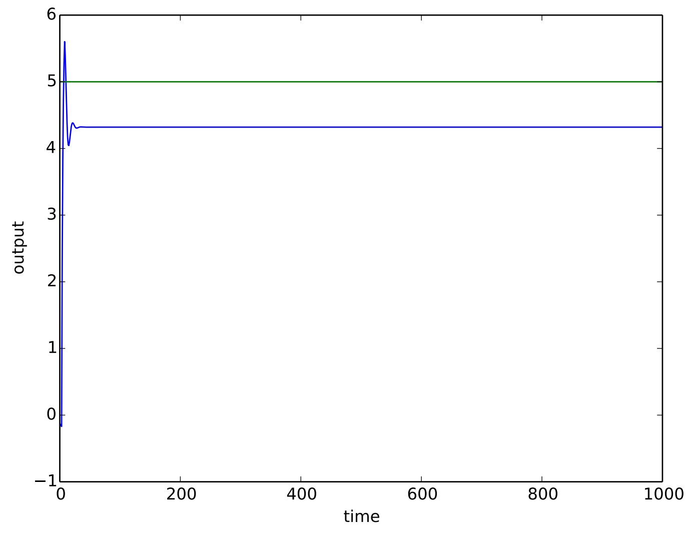
Although training on RandomJump outperforms SineAndJump when the target output is the constant 2, LSTMSNN is able to maintain a low OE as the target output increases. And it is more important for a controller to maintain a smaller OE during the process. We think the size and diversity of data set causes the performance difference. SineAndJump allows LSTMSNN to learn the optimal control actions under more system and target outputs, whereas RandomJump cannot. The output response of trained LSTMSNN model for constant set point of 5 and 10 are shown in Figure 3.8 and 3.9

Table 3.4: Performance comparison between methods

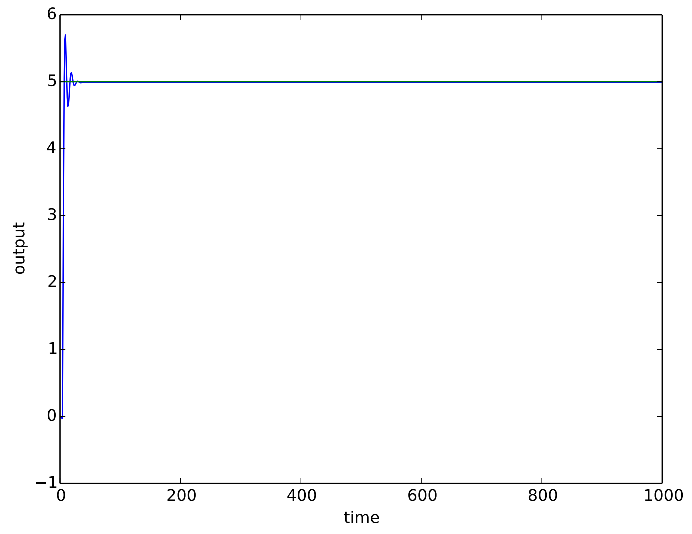
Model	Target Output	MSE	OE
LSTMSNN	2	0.06	0.01
NN-only	2	0.07	0.23
LSTM-only	2	5.56	Did not converge
LSTMSNN	5	0.078	0.01
NN-only	5	0.53	0.7
LSTM-only	5	62.42	Did not converge
LSTMSNN	10	0.01	0.01
NN-only	10	2.21	1.3
LSTM-only	10	167.78	Did not converge

Our second set of experiments shows the effectiveness of LSTMSNN by comparing with NN-only and LSTM-only. In experiment, each model starts with the same initial condition and receives a system output after making a decision on control action in each time step. The current system output, fixed target output and current control action are the input to LSTMSNN and LSTM-only in next time step. The current system output and the fixed target output are the input to NN-only in the next time step, because NN outputs the next control action without using past control action according to Section 3.3.2. We run this process for 1000 time steps for each model. We use SineAndJump to train each model. Results are shown in Table 3.4

Figures 3.8 and 3.9 show the system outputs from LSTMSNN and NN-only. NN-only has a constant gap between its system output and the target output, whereas LSTMSNN does not. The difference in OE between LSTM-

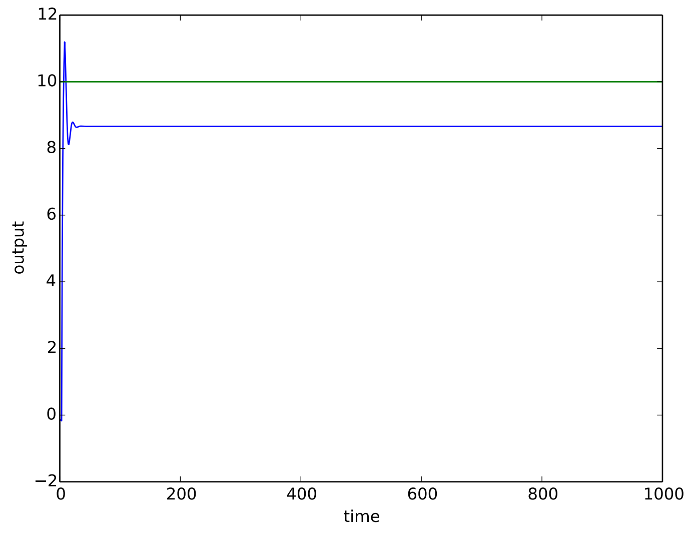


NN-Model controlling the system with for set point = 5

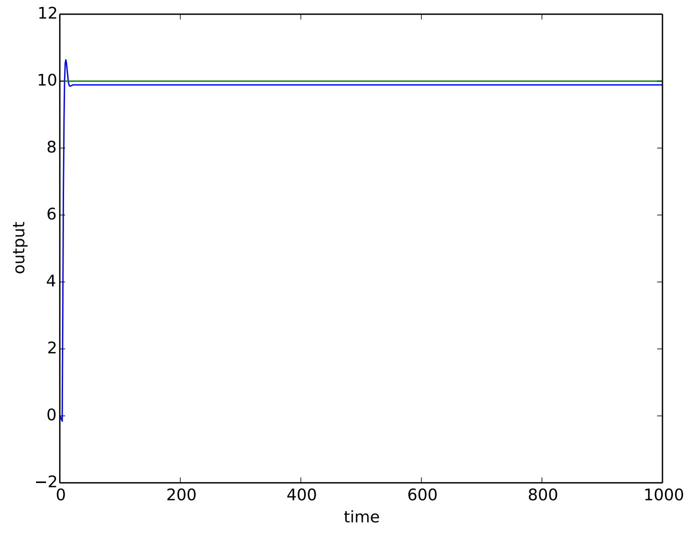


LSTMSNN-Model controlling the system with for set point = 5

Figure 3.8: Comparison of NN only model and LSTMSNN model to track a set point of 5



NN-Model controlling the system with for set point = 10. The set point = 10 was not shown to the model during training



LSTMSNN-Model controlling the system with for set point = 10. The set point = 10 was not shown to the model during training

Figure 3.9: Comparison of NN only model and LSTMSNN model to track a set point of 10

SNN and NN-only also reflects the flaw of NN-only. It is very promising that LSTMSNN maintains an OE close to 0.01 as time step increases. LSTMSNN performs much better than other models in all other cases. A target output of 10 is completely out of the range in our training data, but LSTMSNN still can produce system outputs with low MSE and OE. The reason behind that is LSTMSNN takes control actions calculating the trend of the past control actions using LSTM as well as present system output with NN. The weighted combination of LSTM and NN in LSTMSNN has learnt the time series and mapping (map from output to control actions) trend so well that even though the data set range is -3 to $+3$ it is able to make the system output reach target output of 10 with less oscillatory effect.

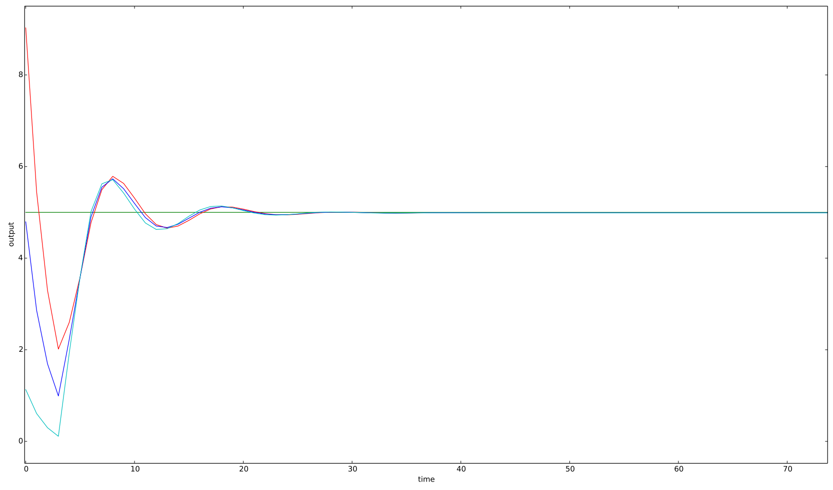
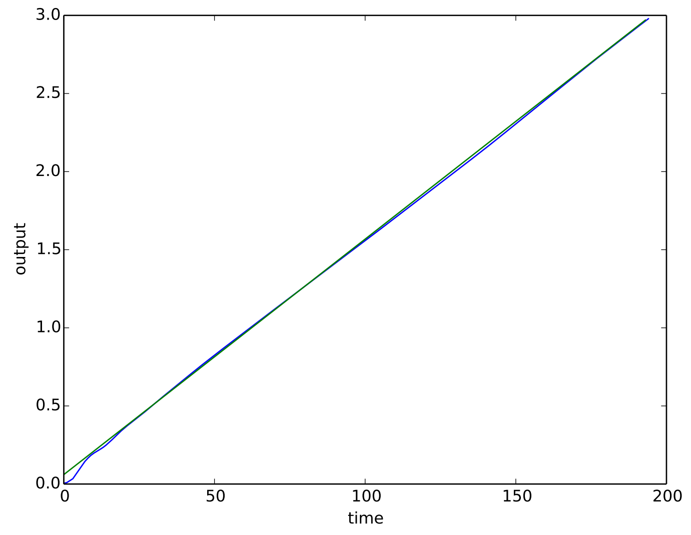
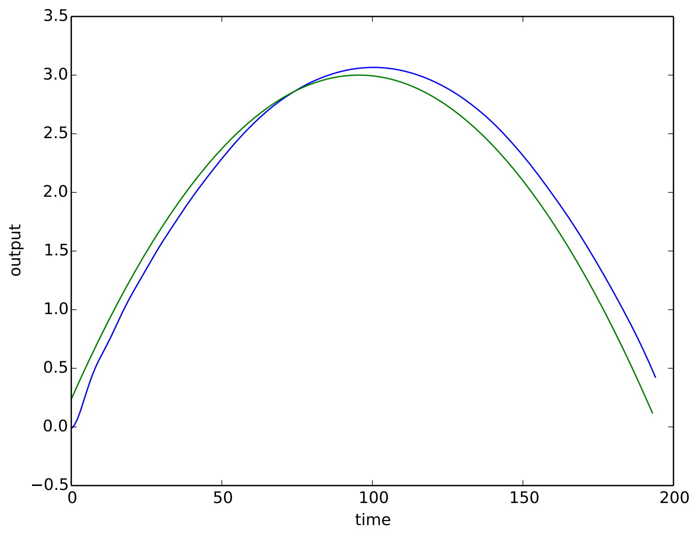


Figure 3.10: LSTMSNN's system output under various initial condition. Green line is the target output. Other colors denote different system outputs under different initial conditions.

The third set of experiments shows LSTMSNN's ability to handle various initial conditions and varying target outputs. We use SineAndJump to train LSTMSNN. We give LSTMSNN a set of random initial conditions and set a fixed target output. From Figure 3.10, we can see that LSTMSNN is able to adjust and produce system output close to the target output after some time steps. We also give LSTMSNN varying target outputs, and we can observe that LSTMSNN can accurately follow the varying target output from Figure 3.11 and 3.12. This demonstrates that LSTMSNN is a robust model. System outputs of LSTMS are very good when the varying target output is smooth. We did find that LSTMSNN's system outputs are not as good when the target output has some sharp corners. In those cases, LSTMSNN has poor performances at those sharp corners but it starts to perform well when target output becomes smooth again.

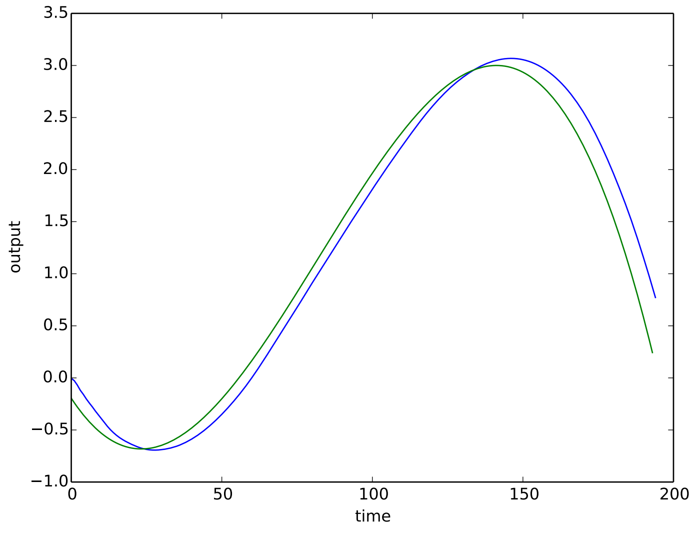


Trained LSTMSNN model tracking a linear set point change

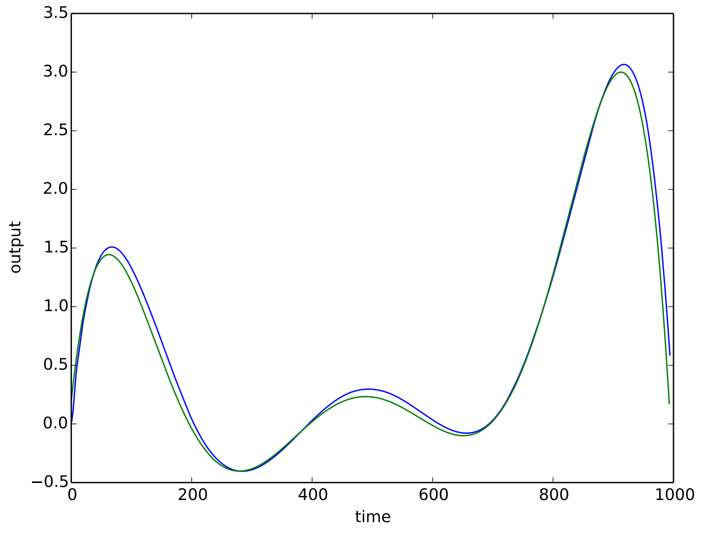


Trained LSTMSNN model tracking a quadratic setpoint change

Figure 3.11: LSTMSNN's system output under varying target output. Green line is target output. Blue line is LSTMSNN's system output.



Trained LSTMSNN model tracking a cubic set point change



LSTMSNN-Model controlling the system with for set point = 10. The set point = 10 was not shown to the model during training

Figure 3.12: LSTMSNN's system output under varying target output. Green line is target output. Blue line is LSTMSNN's system output.

3.6 Conclusion

We have developed a novel neural network architecture to learn the complex behaviour of MPC. This approach eliminates the burden of state estimation, optimization and prediction in MPC, since they are learnt as an abstract information in the hidden layers and hidden units of the network. Also, since our implementation uses the GPU, computing control actions is relatively fast. Hence, the proposed approach can be used to learn the behaviour of MPC offline and the trained model can be deployed in production to control a system. We believe these results are promising enough to justify further research involving complex systems.

Chapter 4

Online Policy Learning using Deep Reinforcement Learning

Referred Conference Publication:
The contributions of this chapter have been published in:
Steven Spielberg Pon Kumar, Bhushan Gopaluni and Philip D. Loewen, *Deep Reinforcement Learning Approaches for Process Control, Advanced Control of Industrial Processes*, Taiwan, 2017
Presentation:
The contributions of this chapter have also been presented in:
Steven Spielberg Pon Kumar, Bhushan Gopaluni, Philip Loewen, *Deep Reinforcement Learning Approaches for Process Control*, American Institute of Chemical Engineers Annual Meeting, San Francisco, USA, 2016.

It is common in the process industry to use controllers ranging from proportional controllers to advanced Model Predictive Controllers (MPC). However, classical controller design procedure involves careful analysis of the process dynamics, development of an abstract mathematical model, and finally, derivation of a control law that meets certain design criteria. In contrast to the classical design process, reinforcement learning offers the prospect of learning appropriate closed-loop controllers by simply interacting with the process and incrementally improving control behaviour. The promise of such an approach is appealing: instead of using a time consuming human-control design process, the controller learns the process behaviour automatically by interacting directly with the process. Moreover, the same underlying learning principle can be applied to a wide range of different process types: linear and nonlinear systems; deterministic and stochastic systems; single input/output and multi input/output systems. From a system identification perspective, both model identification and controller design are performed simultaneously. This specific controller differs from traditional controllers in that it assumes no predefined control law but rather learns the control law from experience. With the proposed approach the reward function serves as an objective function indirectly. The drawbacks of standard control algorithms are: (i) a reasonably accurate dynamic model of the complex process is required, (ii) model maintenance is often very difficult, and (iii) online adaptation is rarely achieved. The proposed reinforcement learning controller is an efficient alternative to standard algorithms and it automatically allows for continuous online tuning of the controller.

The main advantages of the proposed algorithm are as follows: (i) it does not involve deriving an explicit control law; (ii) it does not involve deriving first principle models as they are learnt by simply interacting with the environment; and (iii) the learning policy with deep neural networks is rather fast.

The chapter is organized as follows: Section 4.2 highlights the system configuration in a reinforcment learning perspective. Section 4.3 outlines the technical approach of the learning controller. Section 4.4 empirically verifies the effectiveness of our approach. Conclusions follows in Section 4.5.

4.1 Related Work

Reinforcement learning (RL) has been used in process control for more than a decade [10]. However, the available algorithms do not take into account the improvements made through recent progress in the field of artificial intelligence, especially deep learning [1]. Remarkably, human level control has been attained in games [2] and physical tasks [3] by combining deep learning and reinforcement learning [2]. Recently, these controllers have even learnt the control policy of the complex game of Go [15]. However, the current literature is primarily focused on games and physical tasks. The objective of these tasks differs slightly from that of process control. In process control, the task is to keep the outputs close to a prescribed set point while meeting certain constraints, whereas in games and physical tasks the objective is rather generic. For instance: in games, the goal is to take an action to eventually win a game; in physical tasks, to make a robot walk or stand. This paper discusses how the success in deep reinforcement learning can be applied to process control problems. In process control, action spaces are continuous and reinforcement learning for continuous action spaces has not been studied until [3]. This work aims at extending the ideas in [3] to process control applications.

4.2 Policy Representation

A policy is RL agent's behaviour. It is a mapping from a set of States S to a set of Actions A, i.e., $\pi : S \mapsto A$. We consider a deterministic policy, with both states and actions in a continuous space. The following subsections provide further details about the representations.

4.2.1 States

A state s consists of features describing the current state of the plant. Since the controller needs both the current output of the plant and the required set point, the state is the plant output and set point tuple $< y, y_{set} >$.

4.2.2 Actions

The action, a is the means through which a RL agent interacts with the environment. The controller input to the plant is the action.

4.2.3 Reward

The reward signal r is a scalar feedback signal that indicates how well an RL agent is doing at step t. It reflects the desirability of a particular state transition that is observed by performing action a starting in the initial state s and resulting in a successor state s'. Figure 4.1 illustrates state-action transitions and corresponding rewards of the controller in a reinforcement learning framework. For a process control task, the objective is to drive the output toward the set point, while meeting certain constraints. This specific objective can be fed to the RL agent (controller) by means of a reward function. Thus, the reward function serves as a function similar to an objective function in a Model Predictive Control formulation.

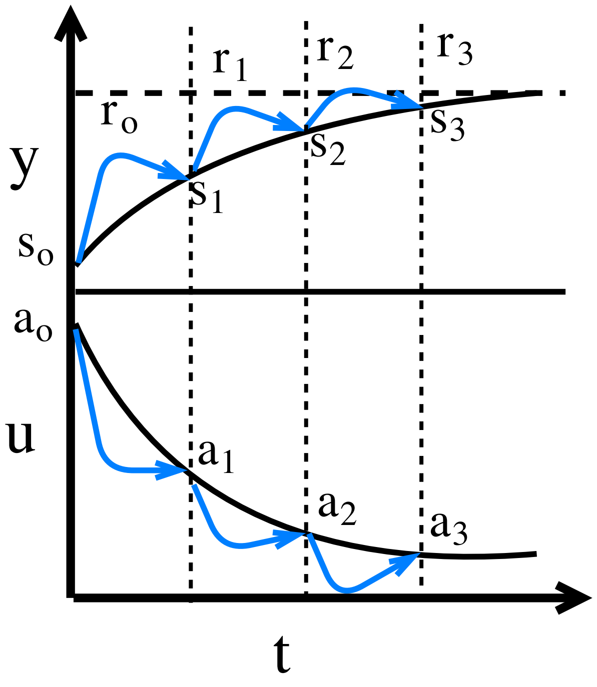


Figure 4.1: Transition of states and actions

The reward function has the form,

$$r(s,a,s') = \begin{cases} c, & \text{if } |y^i - y^i_{set}| \leq \varepsilon, \forall i \\ -\sum_{i=1}^{n} |y^i - y^i_{set}|, & \text{otherwise} \end{cases} \tag{4.1}$$

where i represents i^{th} component in a vector of n outputs in MIMO systems, and $c > 0$ is a constant. A high value of c leads to larger value of r when the outputs are within the prescribed radius ε of the setpoint. A high value of c typically results in quicker tracking of set point.

The goal of learning is to find a control policy, π that maximizes the expected value of the cumulative reward, R. Here, R can be expressed as the time-discounted sum of all transition rewards, r_i, from the current action up to a specified horizon T (where T may be infinite) i.e.,

$$R(s_0) = r_0 + \gamma r_1 + ... + \gamma^T r_T \tag{4.2}$$

where $r_i = r(s_i, a_i, s_i')$ and $\gamma \in (0, 1)$ is a discount factor. The sequence of states and actions are determined by the policy and the dynamics of the system. The discount factor ensures that the cumulative reward is bounded, and captures the fact that events occurring in the distant future are likely to be less consequential than those occurring in the more immediate future. The goal of RL is to maximize the expected cumulative reward.

4.2.4 Policy and Value Function Representation

The policy π is represented by an actor using a deep feed forward neural network parameterized by weights W_a. Thus, an actor is represented as $\pi(s, W_a)$. This network is queried at each time step to take an action given the current state. The value function is represented by a critic using another deep neural network parameterized by weights W_c. Thus, critic is represented as $Q(s, a, W_c)$. The critic network predicts the Q-values for each actor and actor network proposes an action for the given state. During the final runtime, the learnt deterministic policy is used to compute the control action at each step given the current state, s which is a function of current output and set point of the process.

4.3 Learning

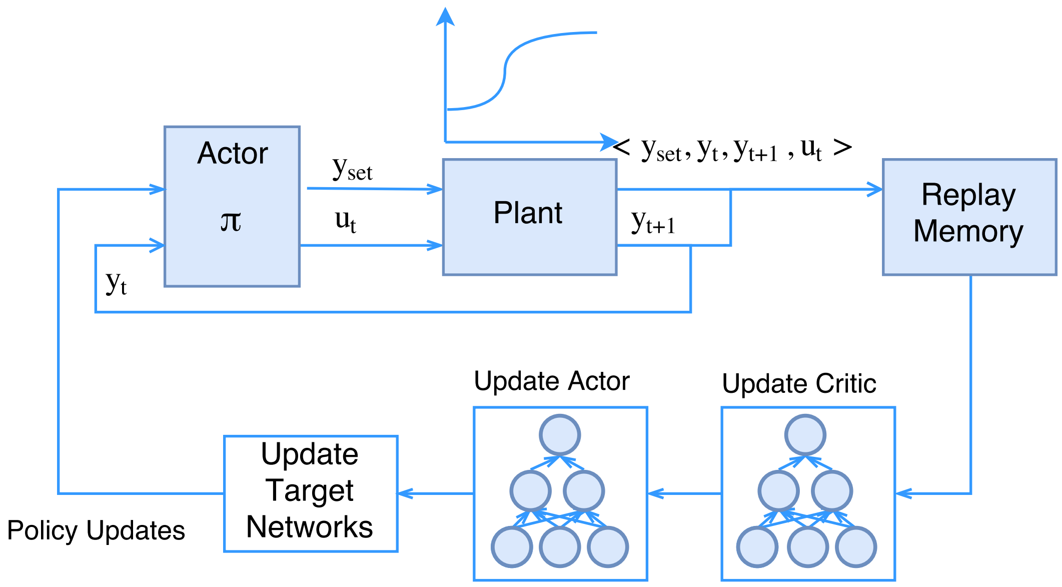


Figure 4.2: Learning Overview

The overview of control and learning of the control system is shown in Fig 4.2. The algorithm for learning the control policy is given in Algorithm 1. The learning algorithm is inspired by [3] with modifications to account for set point tracking and other recent advances discussed in [4],[5]. In order to facilitate exploration, we added noise sampled from an Ornstein-Uhlenbeck (OU) process [14] discussed similarly in [3]. Apart from exploration noise, the random initialization of system output at the start of each episode ensures that the policy is not stuck in a local optimum. An episode is terminated when it runs for 200 time steps or if it has tracked the set point by a factor of ε for five consecutive time steps. For systems with higher values of the time constant, τ, larger time steps per episode are preferred. At each time step the actor, $\pi(s, W_a)$ is queried and the tuple $< s_i, s'_i, a_i, r_i >$ is stored in the replay memory, RM at each iteration. The motivation for using replay memory is to break the correlated samples obtained used for training. The learning algorithm uses an actor-critic framework. The critic network estimates the value of current policy by Q-learning. Thus the critic provides loss function for learning the actor. The loss function of the critic is given by, $L(W_c) = \mathbb{E}\left[(r + \gamma Q_t - Q(s, a, W_c))^2\right]$. Hence, the critic network is updated using this loss with the gradient given by

$$\frac{\partial L(W_c)}{\partial W_c} = \mathbb{E}\left[(r + \gamma Q_t - Q(s, a, W_c))\frac{\partial Q(s, a, W_c)}{\partial W_c}\right], \tag{4.3}$$

where $Q_t = Q(s', \pi(s', W_a^t), W_c^t)$, denotes target values and W_a^t, W_c^t are the weights of the target actor and the target critic respectively and $\frac{\partial Q(s,a,W_c)}{\partial W_c}$ is the gradient of critic with respect to critic parameter W_c. The discount factor $\gamma \in (0, 1)$ determines the present value of future rewards. A value of $\gamma = 1$ considers all rewards from future states to be equally influential whereas a value of 0 ignores all reward except for the current state. Rewards in future states are discounted by value of γ. This is a tunable parameter that makes our learning controller to check how far to look in the future. We chose $\gamma = 0.99$ in our simulations. The expectation in (4.3) is over the mini-batches sampled from RM. Thus the learnt value function provides a loss function to the actor and the actor updates its policy in a direction that improves Q. Thus, the actor network is updated using the gradient given by

$$\frac{\partial J(W_a)}{\partial W_a} = \mathbb{E}\left[\frac{\partial Q(s, a, W_c)}{\partial a}\frac{\partial \pi(s, W_a)}{\partial W_a}\right], \tag{4.4}$$

Algorithm 2 Learning Algorithm

```
1: $W_a, W_c \leftarrow$ initialize random weights
2: initialize Replay memory, RM, with random policies
3: **for** episode = 1 to E **do**
4:     Reset the OU process noise $\mathcal{N}$
5:     Specify set point, $y_{set}$ at random
6:     **for** step = 1 to T **do**
7:         $s \leftarrow < y_t, y_{set} >$
8:         $a \leftarrow$ action, $u_t = \pi(s, W_a) + \mathcal{N}_t$
9:         Execute action $u_t$ on the plant
10:        $s' \leftarrow < y_{t+1}, y_{set} >$, $r \leftarrow$ reward; state & reward at next instant
11:        Store the tuple $< s, a, s', r >$ in RM
12:        Sample a mini batch of n tuples from RM
13:        Compute $y^i = r^i + \gamma Q_t^i \;\; \forall i \in$ mini batch
14:        Update Critic:
15:        $W_c \leftarrow W_c + \alpha(\frac{1}{n}\sum_i^n (y^i - Q(s^i, a^i, W_c)\frac{\partial Q(s^i,a^i,W_c)}{\partial W_c})$
16:        Compute $\nabla_p^i = \frac{\partial Q(s^i,a^i,W_c)}{\partial a^i}$
17:        Clip gradient $\nabla_p^i$ by (4.5)
18:        Update Actor:
19:        $W_a \leftarrow W_a + \alpha\frac{1}{n}\sum_1^n (\nabla_p^i \frac{\partial \pi(s^i,W_a)}{\partial W_a})$
20:        Update Target Critic:
21:        $W_a^t \leftarrow \tau W_a + (1-\tau)W_a^t$
22:        Update Target Actor:
23:        $W_c^t \leftarrow \tau W_c + (1-\tau)W_c^t$
24:     **end for**
25: **end for**
```

where $\frac{\partial Q(s,a,W_c)}{\partial a}$ is the gradient of the critic with respect to the sampled actions of mini-batches from RM and $\frac{\partial \pi(s,W_a)}{\partial W_a}$ is the gradient of actor with respect to parameter, W_a of the actor. Thus, the critic and actor are updated in each iteration, resulting in policy improvement. The layers of actor and critic neural networks are batch normalized [10].

Target Network: Similar to [3], we use separate target networks for actor and critic. We freeze the target network and replace it with the existing network once every 500 iterations. This makes the learning an off-policy algorithm.

Prioritized Experience Replay: While drawing samples from the replay buffer, RM we use prioritized experienced replay as proposed in [4]. Empirically, we found that this accelerates learning compared to uniform random sampling.

Inverting Gradients: Sometimes the actor neural network, $\pi(s, W_a)$ produces an output that exceeds the action bounds of the specific system. Hence, the bound on the output layer of the actor neural network is specified by controlling the gradient required for actor from critic. To avoid making the actor return outputs that violate control constraints, we inverted the gradients of $\frac{\partial Q(s,a,W_c)}{\partial a}$ given in [4] using the following transformation,

$\nabla_p =$

$$\nabla_p . \begin{cases} (p_{max} - p)/(p_{max} - p_{min}), & \text{if } \nabla_p \text{suggests increasing } p \\ (p - p_{min})/(p_{max} - p_{min}), & \text{otherwise} \end{cases} \tag{4.5}$$

Where ∇_p refers to parameterized gradient of the critic. Here p_{max}, p_{min} set the maximum and minimum actions of the agent. p correspond to entries of the parameterized gradient, $\frac{\partial Q(s,a,W_c)}{\partial a}$. This tranformation makes sure that the gradients are clipped to truncate the output of actor network i.e., control input, within the range given in p_{min} and p_{max}, where p_{min} and p_{max} are lower and upper bound on the action space. It serves as an input constraint to the RL agent.

4.4 Implementation and Structure

The learning algorithm was written in python and implemented on a Ubuntu linux distribution machine. The Random Access Memory of the machine is 16 GB. The deep neural networks was built using Tensorflow [41]. The computation of high dimensional matrix multiplication was made parallel with the help of Graphics Processing Unit (GPU) and we used NVIDIA 960M GPU.

The details of our actor and critic neural networks are as follows: We used Adam [21] for learning actor and critic neural network. The learning rate used for actor was 10^{-4} and critic was 10^{-3}. For critic we added L^2 regularization with penalty $\lambda = 10^{-2}$. The discount factor to learn critic was $\gamma = 0.99$. We used Rectified non-linearity as an activation function for all hidden layers in actor and critic. The actor and critic neural networks had 2 hidden layers with 400 and 300 units respectively. The Adam optimizer was trained in minibatch sizes of 64. The size of the replay buffer is 10^4.

4.5 Simulation Results

We tested our learning based controller on a paper-making machine (Single Input Single Output system) and a high purity distillation column (Multi Input Multi Output System). The output and input responses from the learned polices are illustrated in Fig. 4.3 - 4.12. The results are discussed below.

4.5.1 Example 1- SISO Systems

A 1×1 process is used to study the effect of this approach of learning the policy. The industrial system we choose to study is the control of a paper-making machine. The target output, y_{set}, is the desired moisture content of the paper sheet. The control action, u, is the steam flow rate, and the system output, y, is the current moisture content. The differential equation of the system is given by,

$$y(t) - 0.6\dot{y}(t) = 0.05\dot{u}(t), \tag{4.6}$$

where $\dot{y}(t)$ and $\dot{u}(t)$ are output and input derivatives with respect to time. The corresponding transfer function of the system is,

$$G(s) = \frac{0.05s}{1 - 0.6s} \tag{4.7}$$

The time step used for simulation is 1 second.
The system is learned using Algorithm 1. The portion of the learning curve of the SISO system given in (4.6) is shown in Fig 4.3 and the corresponding learned policy is shown in Fig 4.4.

The learned controller is also tested on various other cases, such as the response to a set point change and the effect of output and input noise. The

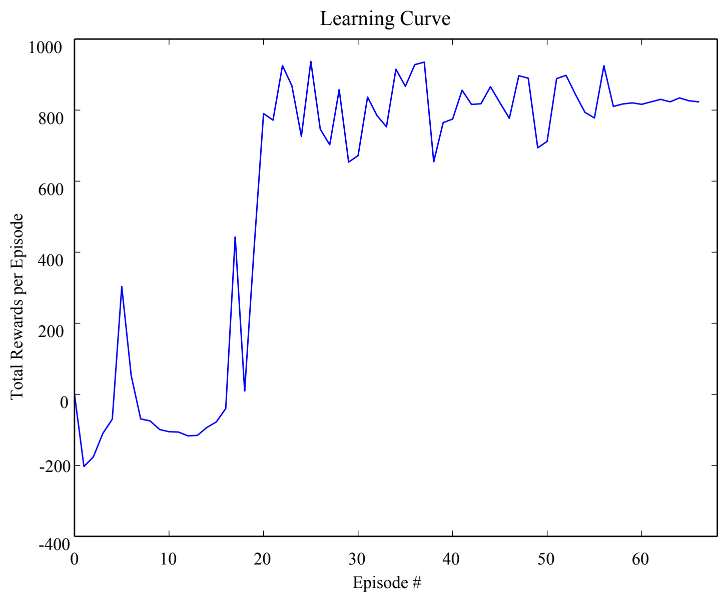
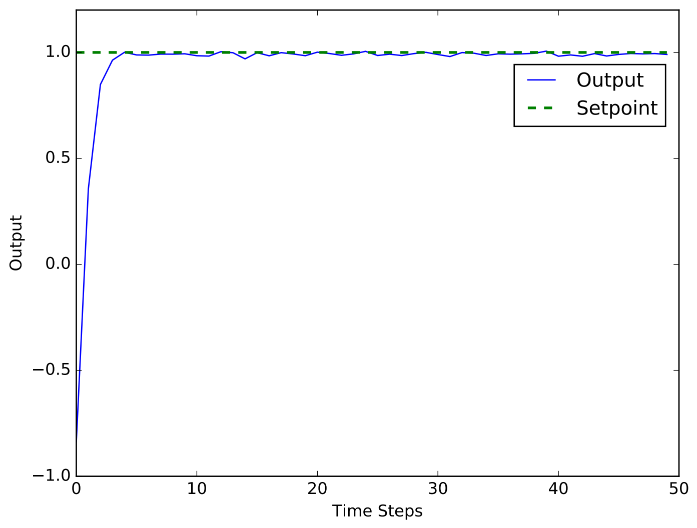
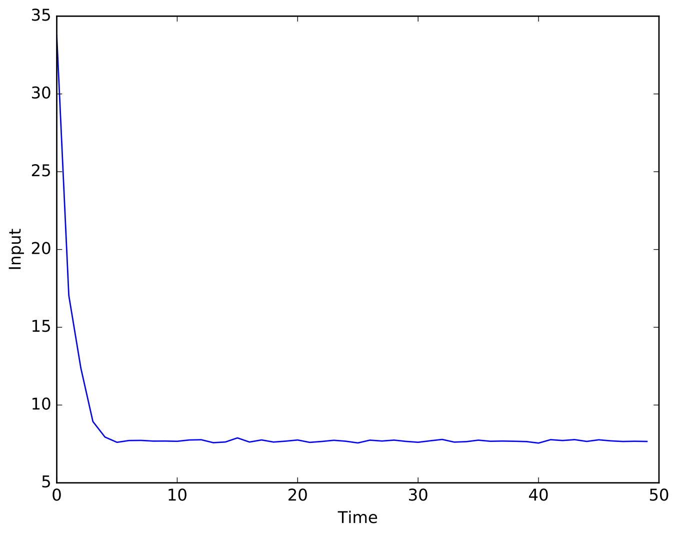


Figure 4.3: Learning Curve: Rewards to the Reinforcement Learning agent for system (4.6) over time. Here 1 episode is equal to 200 time steps.

results are shown in Fig 4.5 - Fig 4.10 . For learning set point changes, the learning algorithm was shown integer-valued set points in the range $[0, 10]$.



Output response of the learnt policy



Input response of the learnt policy

Figure 4.4: Output and Input response of the learnt policy for system (4.6)

It can be seen from Fig 4.10 that the agent has learned how to track even set points that it was not shown during training.

4.5.2 Example 2- MIMO System

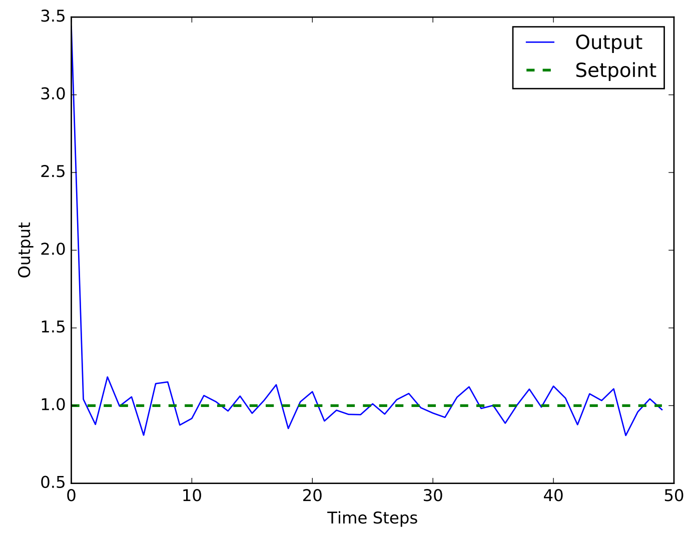
Our approach is tested on a high purity distillation column Multi Input Multi Output (MIMO) system described in [57]. The first output, y_1, is the distillate composition, the second output, y_2, is the bottom composition, the first control input, u_1, is the boil-up rate and the second control input, u_2, is the reflux rate. The differential equations of the MIMO is,

$$\begin{aligned} \tau \dot{y_1}(t) + y_1(t) &= 0.878u_1(t) - 0.864u_2(t) \\ \tau \dot{y_2}(t) + y_2(t) &= 1.0819u_1(t) - 1.0958u_2(t) \end{aligned} \tag{4.8}$$

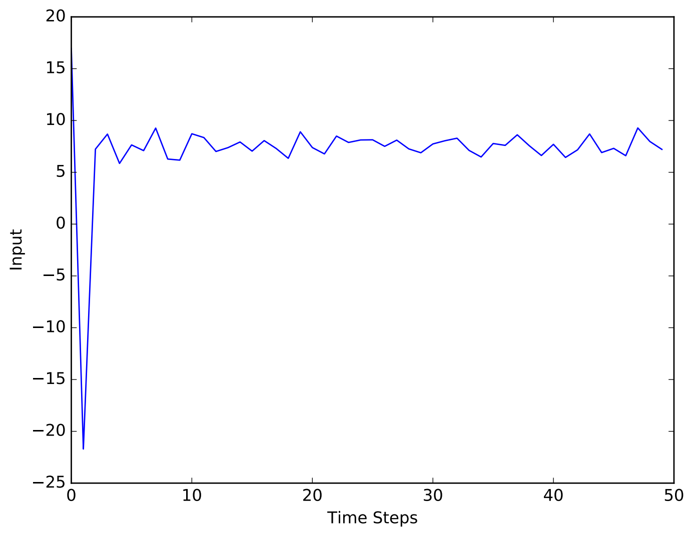
where $\dot{y_1}(t)$ and $\dot{y_1}(t)$ are derivatives with respect to time. The output and input responses of the learned policy of the system are shown in Fig 4.11 and Fig 4.12

We choose the number of steps per episode to be 200 for learning the control policies. When systems with larger time constants are encountered, we usually require more steps per episode. The reward hypothesis formulation serves as a tuner to adjust the performance of the controller, how set points should be tracked, etc. When implementing this approach on a real plant, effort can often be saved by warm-starting the deep neural network through prior training on a simulated model. Experience also suggests using exploration noise with higher variance in the early phase of learning, to encourage high exploration.

The final policies for the learned SISO system, given in (4.6), are the result of 7,000 iterations of training collecting about 15,000 tuples and requiring

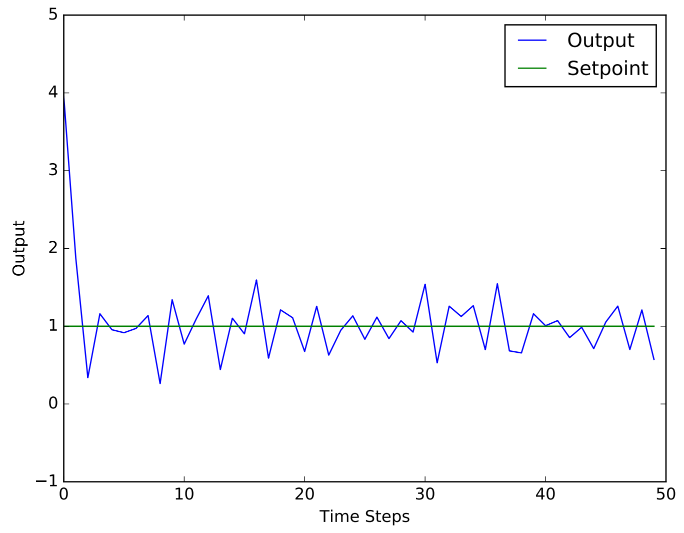


Output response of the learnt system with output noise of $\sigma^2 = 0.1$

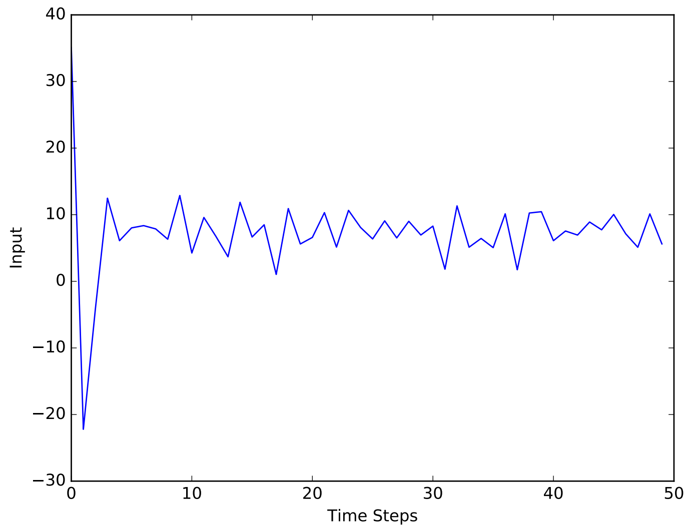


Input response of the learnt system with output noise of $\sigma^2 = 0.1$

Figure 4.5: Output and input response of the learnt system with output noise of $\sigma^2 = 0.1$ for SISO system (4.6)

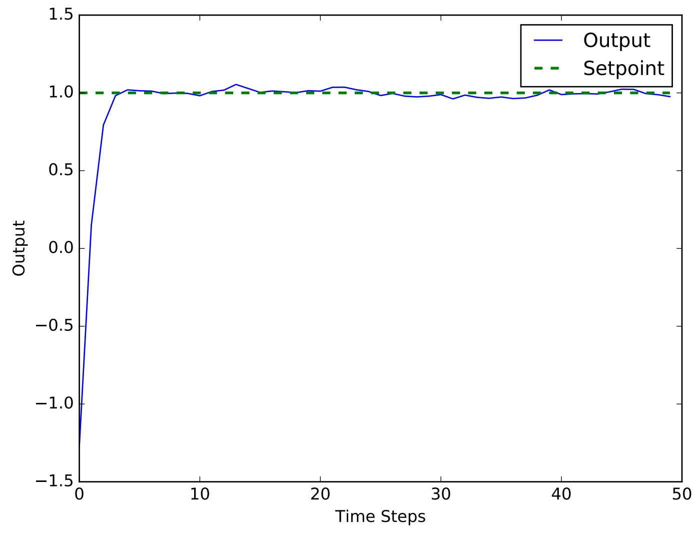


Output response of the learnt system with output noise of $\sigma^2 = 0.3$

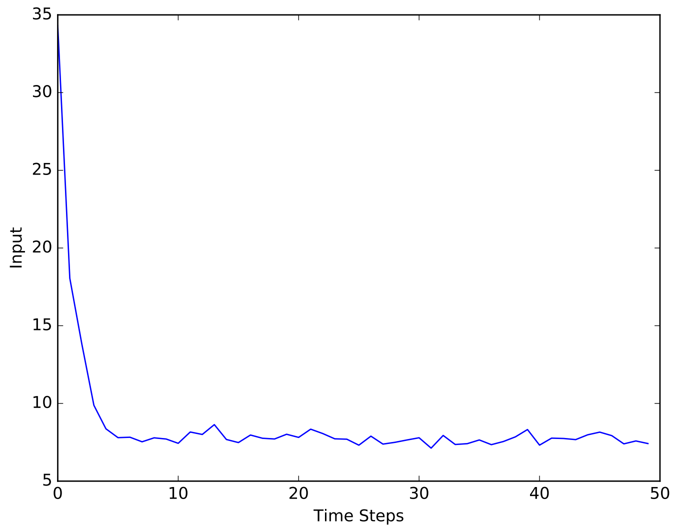


Input response of the learnt system with output noise of $\sigma^2 = 0.3$

Figure 4.6: Output and input response of the learnt system with output noise of $\sigma^2 = 0.3$ for SISO system (4.6)

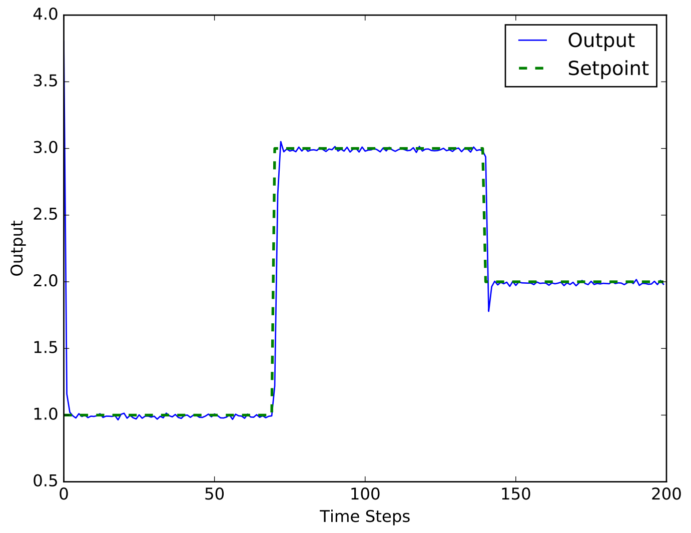


Output response of the learnt system with input noise of $\sigma^2 = 0.1$

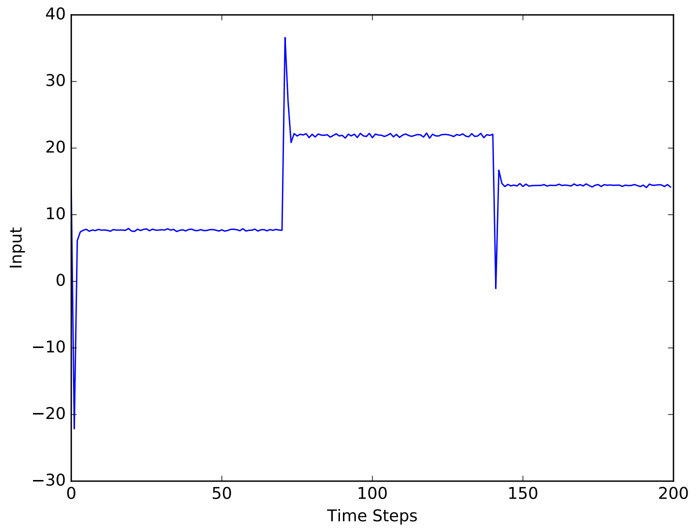


Input response of the learnt system with input noise of $\sigma^2 = 0.1$

Figure 4.7: Output and input response of the learnt system with input noise of $\sigma^2 = 0.1$ for SISO system (4.6)

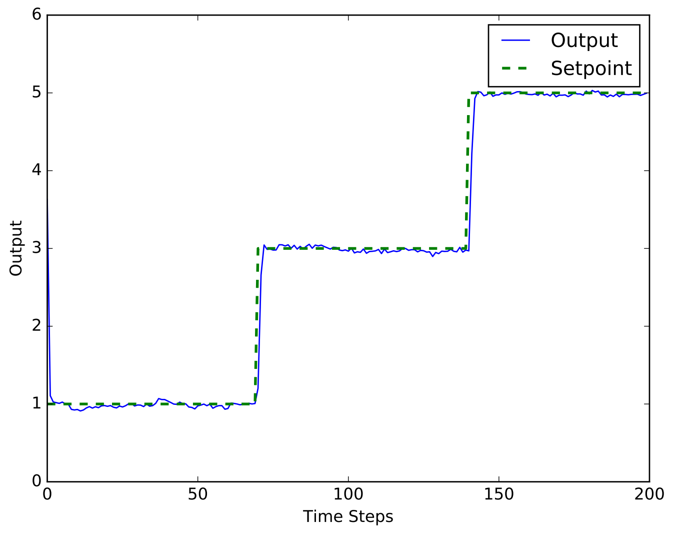


Output response of the learnt system with set point change and output noise of $\sigma^2 = 0.1$

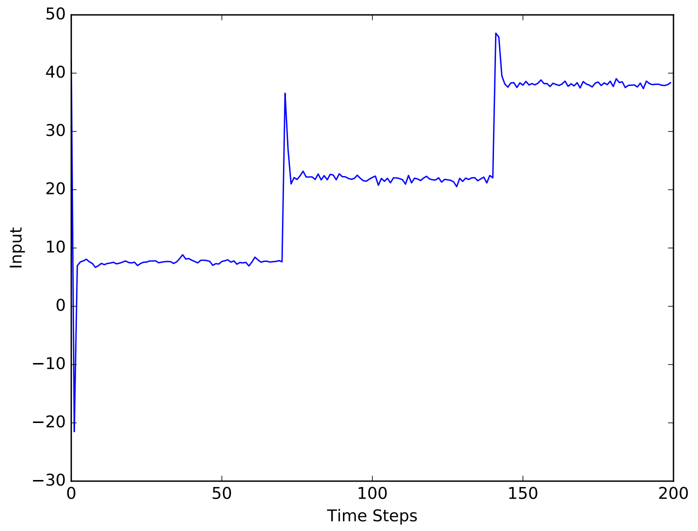


Input response of the learnt system with set point change and output noise of $\sigma^2 = 0.1$ for system (4.6)

Figure 4.8: Output and input response of the learnt SISO system with set point change and output noise of $\sigma^2 = 0.1$

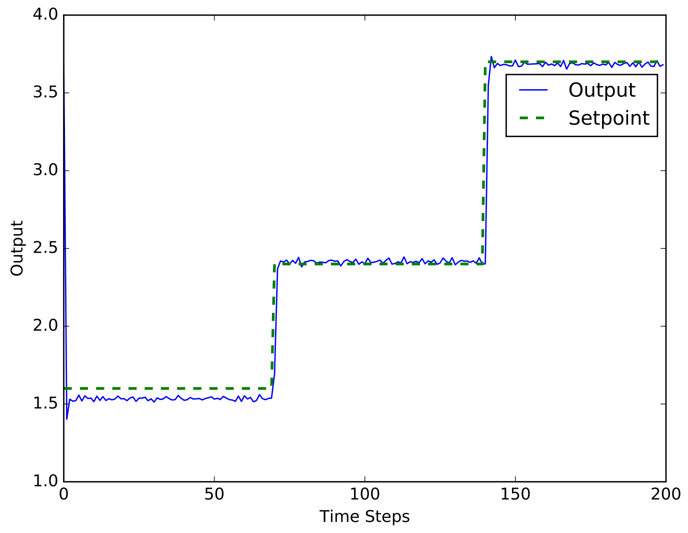


Output response of the learnt system with set point change and output and input noise of $\sigma^2 = 0.1$

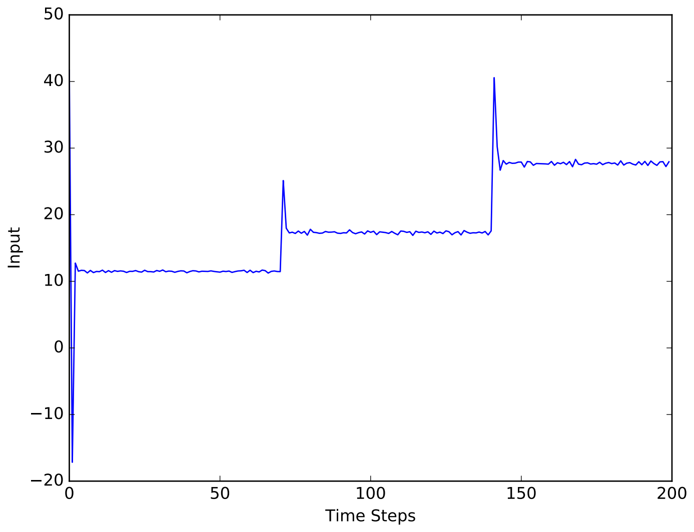


Input response of the learnt system with set point change and output and input noise of $\sigma^2 = 0.1$

Figure 4.9: Output and input response of the learnt system with set point change and output and input noise of $\sigma^2 = 0.1$ for SISO system (4.6)

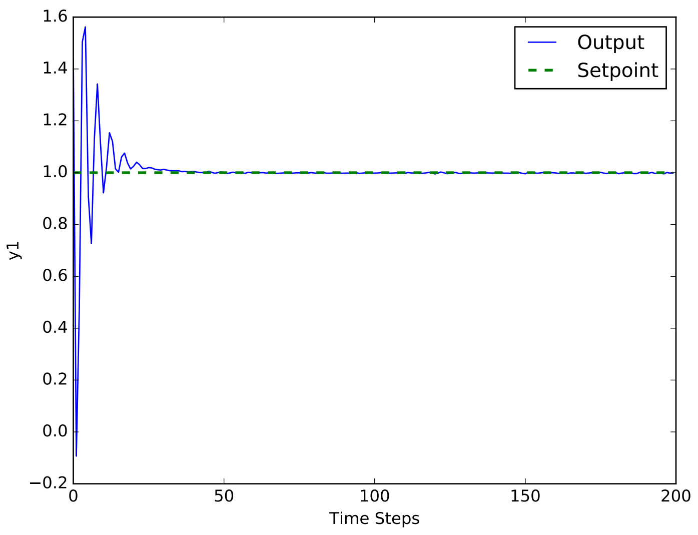


Output response of the learnt system for set points unseen during training

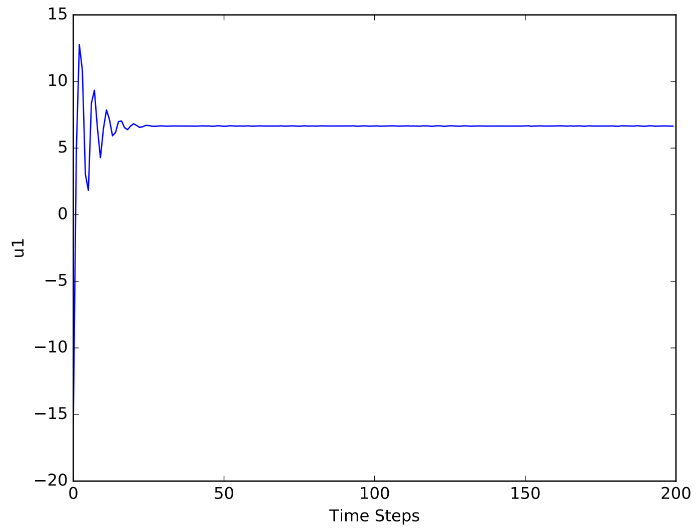


Input response of the learnt system for set points unseen during training

Figure 4.10: Output and input response of the learnt SISO system (4.6) for set points unseen during training

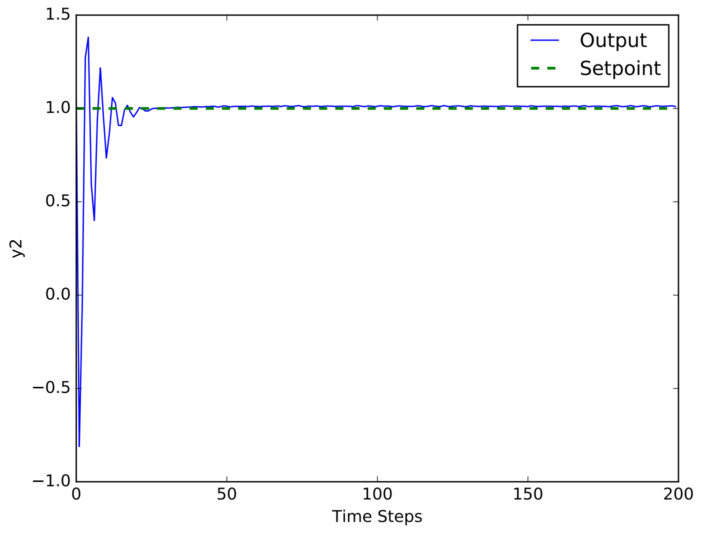


Output response of the first variable with output noise of $\sigma^2 =$ 0.01 of the 2×2 MIMO system

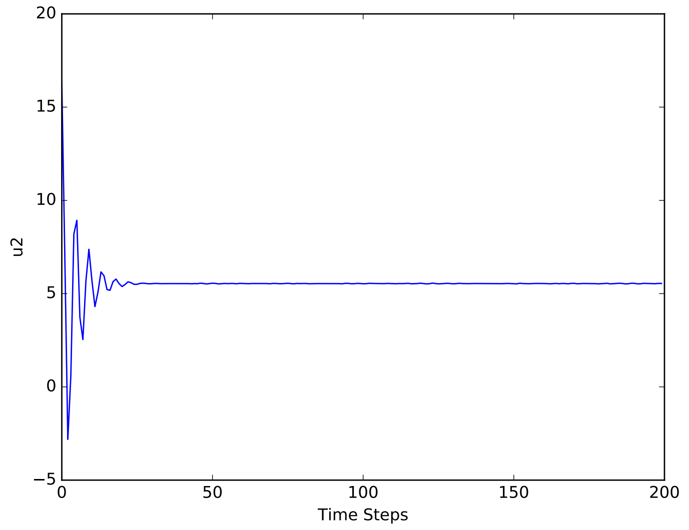


Input response of the first variable with output noise of $\sigma^2 =$ 0.01 of the 2×2 MIMO system

Figure 4.11: Output and input response of the first variable with output noise of $\sigma^2 = 0.01$ of the 2×2 MIMO system (4.8)



Output response of the second variable with output noise of $\sigma^2 = 0.01$ of the 2×2 MIMO system



Input response of the second variable with output noise of $\sigma^2 = 0.01$ of the 2×2 MIMO system

Figure 4.12: Output and input response of the second variable with output noise of $\sigma^2 = 0.01$ of the 2×2 MIMO system (4.8)

about 2 hours of training on a NVIDIA 960M GPU. For learning the MIMO system given in (4.8) the compute time on the same GPU was about 4 hours requiring 50K tuples. The learning time remains dominated by the mini-batch training at each time step.

4.6 Conclusion

The use of deep learning and reinforcement learning to build a learning controller was discussed. We believe process control will see development in the area of reinforcement learning based controller.

Chapter 5

Conclusion & Future Work

We have developed an artificial intelligence based approach to process control using deep learning and reinforcement learning. We posed our problem as supervised learning as well as a reinforcement learning based problem. This framework supports the development of control policies that can learn directly by interacting with plant's output. All our work, just requires plant output to return optimal control actions, hence these are relatively fast compared to currently available controllers. Our approach avoids the need for hand-crafted feature descriptors, controller tuning, deriving control laws, and developing mathematical models. We believe industrial process control will see rapid and significant advances in the field of deep and reinforcement learning in the near future. Possibilities for future work include the following,

MPC + Reinforcement Learning In Chapter III we combined MPC with deep learning to build a controller that can learn offline and then be able to control the plant. Our future work aims to learn the MPC and plant online. MPC relies on an accurate model of the plant for best performance. Hence with this approach the reinforcement learning applied to MPC and plant will help to adapt to change in the plant and the considered model.

Observation of Hidden states Our new architecture LSTMSNN was used to learn the behaviour of MPC. Our observation to hidden states is that LSTMSNN is learning information about hidden states through available hidden neurons in different levels. It would be instructive to study the information about these hidden units and how the hidden states are learnt, through relevant visualization and theoretical justification.

Bibliography

[1] Krizhevsky, A., I. Sutskever, and G. E. Hinton “Imagenet classification with deep convolutional neural networks”. In Advances in neural information processing systems, Lake Tahoe. pp. 1097-1105, 2012.

[2] Volodymyr, M., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg and D. Hassabis “Human-level control through deep reinforcement learning.” Nature 518, no. 7540: 529-533, 2015.

[3] Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. “Continuous control with deep reinforcement learning.” arXiv preprint arXiv:1509.02971, 2015.

[4] Hausknecht, M., and P. Stone. “Deep reinforcement learning in parameterized action space.” arXiv preprint arXiv:1511.04143, 2015.

[5] Schaul, T., J. Quan, I. Antonoglou, and D. Silver. “Prioritized experience replay.” arXiv preprint arXiv:1511.05952, 2015.

[6] NVIDIA, NVIDIA CUDA™Programming Guide, December 2008.

[7] Hinton, G., L. Deng, D. Yu, G. E. Dahl, A. R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups." IEEE Signal Processing Magazine 29, no. 6: 82-97, 2012.

[8] Yao K., G. Zweig, M. Y. Hwang, Y. Shi, and D. Yu. "Recurrent neural networks for language understanding." Interspeech, Lyon, France, pp. 2524-2528, 2013.

[9] Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602, 2013.

[10] Hoskins, J. C., and D. M. Himmelblau "Process control via artificial neural networks and reinforcement learning". Computers & chemical engineering, 16(4), 241-251.

[11] Ioffe, S. and C. Szegedy "Batch normalization: Accelerating deep network training by reducing internal covariate shift" arXiv preprint arXiv:1502.03167, 2015.

[12] Silver, D., G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. "Deterministic policy gradient algorithms" Proceedings of the 31st International Conference on Machine Learning, Beijing, China, pp. 387-395, 2014.

[13] Bertsekas, D. P., and J. N. Tsitsiklis. "Neuro-dynamic programming: an

overview." Proceedings of the 34th IEEE Conference on Decision and Control, Louisiana, USA, vol. 1, pp. 560-564. IEEE, 1995.

[14] Uhlenbeck, G. E., and L. S. Ornstein. "On the theory of the Brownian motion." Physical review 36, no. 5 : 823, 1930.

[15] Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. V. D. Driessche,J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe,J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search". Nature. Jan 28;529(7587):484-489, 2016.

[16] Goodfellow, I., Y. Bengio, and A. Courville. Deep learning. MIT press, 2016.

[17] Sutton, R. S., and A. G. Barto. Reinforcement learning: An introduction. Vol. 1, no. 1. Cambridge: MIT press, 1998.

[18] Polyak, B. T., and A. B. Juditsky. "Acceleration of stochastic approximation by averaging." SIAM Journal on Control and Optimization 30, no. 4: 838-855, 1992.

[19] Duchi, J., E. Hazan, and Y. Singer. "Adaptive subgradient methods for online learning and stochastic optimization." Journal of Machine Learning Research 12, no.: 2121-2159, 2011.

[20] Tieleman, T. and G. Hinton, Lecture 6.5 RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012

[21] Kingma, D., and J. Ba. “Adam: A method for stochastic optimization.” arXiv preprint arXiv:1412.6980, 2014.

[22] Sutskever, I., J. Martens, and G. E. Hinton. “Generating text with recurrent neural networks.” In Proceedings of the 28th International Conference on Machine Learning (ICML-11), Washington, USA, pp. 1017-1024, 2011.

[23] Wu, Y., S. Zhang, Y. Zhang, Y. Bengio, and R. R. Salakhutdinov. “On multiplicative integration with recurrent neural networks.” In Advances in Neural Information Processing Systems, Spain, pp. 2856-2864. 2016.

[24] Bengio, Y., P. Simard, and P. Frasconi. “Learning long-term dependencies with gradient descent is difficult”. IEEE Transactions on Neural Networks, 5(2):157-166, 1994.

[25] Pascanu, R., T. Mikolov, and Y. Bengio. ”On the difficulty of training recurrent neural networks.” In International Conference on Machine Learning, Atlanta, USA, pp. 1310-1318. 2013.

[26] Hochreiter, S., and J. Schmidhuber. “Long short-term memory.” Neural computation 9, no. 8: 1735-1780, 1997.

[27] LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. “Gradient-based learning applied to document recognition.” Proceedings of the IEEE 86, no. 11: 2278-2324, 1998.

[28] Bergstra, J., and Y. Bengio. “Random search for hyper-parameter optimization.” Journal of Machine Learning Research 13: 281-305, 2012.

[29] Mayne, D. Q., M. M. Seron, and S. V. Rakovic. "Robust model predictive control of constrained linear systems with bounded disturbances." Automatica 41, no. 2 : 219-224, 2005.

[30] Greff, K., R. K. Srivastava, J. Koutnk, B. R. Steunebrink, and J. Schmidhuber. "LSTM: A search space odyssey." IEEE transactions on neural networks and learning systems, 2017.

[31] Degris, T., M. White, and R. S. Sutton. "Off-policy actor-critic." arXiv preprint arXiv:1205.4839, 2012.

[32] Todorov, E., T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control" IEEE/RSJ International Conference on Intelligent Robots and Systems, Portugal, pp. 5026-5033, 2012

[33] Kuhne, F., W. F. Lages, and J. M. G. D. Silva Jr. "Model predictive control of a mobile robot using linearization." In Proceedings of mechatronics and robotics, pp. 525-530, 2004.

[34] Kittisupakorn, P., P. Thitiyasook, M. A. Hussain, and W. Daosud. "Neural network based model predictive control for a steel pickling process." Journal of Process Control 19, no. 4 : 579-590, 2009.

[35] Piche, S., J. D. Keeler, G. Martin, G. Boe, D. Johnson, and M. Gerules. "Neural network based model predictive control." In Advances in Neural Information Processing Systems, pp. 1029-1035. 2000.

[36] Pan, Y., and J. Wang. "Two neural network approaches to model predictive control." In American Control Conference, Washington, pp. 1685-1690. IEEE, 2008.

[37] Alexis, K., G. Nikolakopoulos, and A. Tzes, "Model predictive quadrotor control: attitude, altitude and position experimental studies". IET Control Theory & Applications, 6(12), pp.1812-1827, 2012.

[38] Wallace, M., S. S. P. Kumar, and P. Mhaskar. "Offset-free model predictive control with explicit performance specification." Industrial & Engineering Chemistry Research 55, no. 4: 995-1003, 2016.

[39] Zhong, M., M. Johnson, Y. Tassa, T. Erez, and E. Todorov. "Value function approximation and model predictive control." IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), pp. 100-107. IEEE, 2013.

[40] Akesson, B. M., and H. T. Toivonen. "A neural network model predictive controller." Journal of Process Control 16 no.9: 937-946, 2006.

[41] Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." arXiv preprint arXiv:1603.04467, 2016.

[42] Spielberg, S. P. K., R. B. Gopaluni, and P. D. Loewen. "Deep reinforcement learning approaches for process control." In Advanced Control

of Industrial Processes (AdCONIP), Taiwan, 6th International Symposium, pp. 201-206. IEEE, 2017.

[43] Hassabis, D., "AlphaGo: using machine learning to master the ancient game of Go", `https://blog.google/topics/machine-learning/alphago-machine-learning-game-go/`, January, 2016

[44] Karpathy, A., "Deep Reinforcement Learning: Pong from Pixels", `http://karpathy.github.io/2016/05/31/rl/`, May 31, 2016

[45] Levy, K., "Here's What It Looks Like To Drive In One Of Google's Self-Driving Cars On City Streets", `http://www.businessinsider.com/google-self-driving-cars-in-traffic-2014-4`, April, 2014.

[46] Welinder, P., B. Mcgrew, J. Schneider, R. Duan, J. Tobin, R. Fong, A. Ray, F. Wolski, V. Kumar, J. Ho, M. Andrychowicz, B. Stadie, A. Handa, M. Plappert, E. Reinhardt, P. Abbeel, G. Brockman, I. Sutskever, J. Clark & W. Zaremba., "Robots that learn", `https://blog.openai.com/robots-that-learn/`, May 16, 2017.

[47] Stelling, S., "Is there a graphical representation of bias-variance tradeoff in linear regression?", https://stats.stackexchange.com/questions/19102/is-there.., November 29, 2011.

[48] Macdonald K., "k-Folds Cross Validation in Python", `http://www.kmdatascience.com/2017/07/k-folds-cross-validation-in-python.html`, July 11, 2017.

[49] Preetham V. V., “Mathematical foundation for Activation Functions in Artificial Neural Networks”, https://medium.com/autonomous-agents/mathematical-foundation-for-activation.., August 8, 2016.

[50] Klimova, E., “Mathematical Model of Evolution of Meteoparameters: Developing and Forecast”, http://masters.donntu.org/2011/fknt/klimova/.., 2011.

[51] Boyan, J. A. “Least-squares temporal difference learning.” International Conference on Machine Learning, Bled, Slovenia, pp. 49-56. 1999.

[52] Camacho, E. F. and C. Bordons, “Model Predictive Control”, Springer Science & Business Media. 1998.

[53] Cho, K., B. V. Merrienboer, D. Bahdanau, and Y. Bengio. “On the properties of neural machine translation: Encoder-decoder approaches.” arXiv preprint arXiv:1409.1259, 2014.

[54] Bojarski, M., D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, K. Zieba. “End to end learning for self-driving cars.” arXiv preprint arXiv:1604.07316, 2016.

[55] Schmidt, M., “Proximal gradient and stochastic sub gradient”, Machine Learning, Fall-2016. https://www.cs.ubc.ca/ schmidtm/Courses/540-W16/L7.pdf

[56] Postoyan, R., L. Busoniu, D. Nesic and J. Daafouz, “Stability Analysis of Discrete-Time Infinite-Horizon Optimal Control With Discounted

Cost," in IEEE Transactions on Automatic Control, vol. 62, no. 6, pp. 2736-2749, 2017.

[57] Patwardhan, R. S., and R. B. Goapluni. "A moving horizon approach to input design for closed loop identification." Journal of Process Control 24.3: 188-202, 2014.

[58] Sutton, R. S., D. A. McAllester, S. P. Singh, and Y. Mansour. "Policy gradient methods for reinforcement learning with function approximation." In Advances in neural information processing systems, Denver, United States, pp. 1057-1063. 2000.

[59] Precup, D., R. S. Sutton, and S. Dasgupta. "Off-policy temporal-difference learning with function approximation." International Conference on Machine Learning, Massachusetts, USA, pp. 417-424. 2001.