



NTNU – Trondheim
Norwegian University of
Science and Technology

Pruning of RBF Networks in Robot Manipulator Learning Control

Siri Vestheim

Master of Science in Engineering Cybernetics

Submission date: June 2012

Supervisor: Jan Tommy Gravdahl, ITK

Co-supervisor: Serge Gale, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Pruning of RBF Networks in Robot Manipulator Learning Control

Master Thesis

Siri Vestheim
Dept. of Engineering Cybernetics, NTNU
`sirives@stud.ntnu.no`

Supervisor: Professor Jan Tommy Gravdahl
Co-Supervisor: Phd. Student Serge Gale

June 8, 2012



Master's Thesis

Student's name: Siri Vestheim

Field: Engineering Cybernetics

Title (Norwegian):

Title (English): Pruning of RBF Networks in Robot Manipulator Learning Control

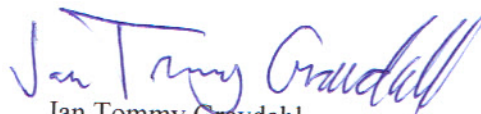
Description:

Neural networks are known to become too large and complex for efficient use in online control of robotic manipulators. This project is concerned with methods for reducing the size of such networks.

Assignments:

1. Give a brief introduction to Radial Basis Function Neural Network
2. Perform a literature review of pruning of Neural Networks
3. Propose at least one pruning method that is suited for pruning of Neural Networks used in online control of robotic manipulators
4. Apply the pruning method(s) to simulations of NN-based online control of
 - a. A 2-DOF planar manipulator
 - b. The three first DOFs of the ABB IRB140
5. Write a paper, based on the main results from the thesis work, to be submitted to the 2013 IEEE International Conference on Robotics and Automation (ICRA)

Advisor(s): Serge Gale, NTNU


Jan Tommy Gravdahl
Supervisor

Abstract

Radial Basis Function Neural Networks are well suited for learning the system dynamics of a robot manipulator and implementation of these networks in the control scheme for a manipulator is a good way to deal with the system uncertainties and modeling errors which often occur. The problem with RBF networks however is to find a network with suitable size, not too computational demanding and able to give accurate approximations. In general two methods for creating an appropriate RBF network has been developed, 1) *Growing* and 2) *Pruning*.

In this report two different pruning methods which are suitable for use in a learning controller for robot manipulators are proposed, *Weight Magnitude Pruning* and *Neuron Output Pruning*. Weight Magnitude Pruning is based on a pruning scheme in [8] while Neuron Output Pruning is based on [2]. Both pruning methods are simple, have low computational cost and are able to remove several units in one pruning period. The thresholds used to find which neurons to remove are specified as a percent and hence less problem dependent to find.

Simulations with the two proposed pruning methods in a learning inverse kinematic controller for tracking a trajectory by using the three first joints of the ABB IRB140 manipulator are conducted. The result was that implementing pruned RBF networks in the controller made it more robust towards system uncertainties due to increased generalization ability. These pruned networks were found to give better tracking in the case of unmodeled dynamics compared to the incorrect system model, not pruning the RBFNNs and a type of growing network called RANEKFs. Computational costs were also reduced when the pruning schemes were implemented.

NTNU has a manipulator of the type ABB IRB140 and the learning inverse kinematic controller with pruning of RBF networks should be implemented and tested on this in real-life simulations.

Sammendrag

Implementering av Radial Basis Funksjon Nevrale Nettverk i en kontroller for å lære systemdynamikken til en robot manipulatorer er en god måte å håndtere systemets usikkerhet og modelleringsfeil som ofte forekommer på. Problemet med RBF nettverk er imidlertid å finne et nettverk med passende størrelse som ikke er altfor beregningskrevende men fremdeles i stand til å gi nøyaktige estimeringer. Det har blitt utviklet to metoder for å lage et passende RBF nettverk, 1) *Voksende* og 2) *Beskjærende*.

I denne rapporten er det utviklet to ulike beskjæringsmetoder som er egnet til bruk i en lærende kontroller for robot manipulatorer. De er kalt Vekt Styrke Beskjæring og Neuron Utgang Beskjæring. Vekt Styrke Beskjæring er basert på en annen beskjæringsmetode i [8] mens Neuron Utgang Beskjæring er basert på [2]. Begge beskjæringsmetodene er enkle og har lave beregningskostnader. De fjerner de fleste av enhetene i løpet av en beskjæringsperiode og gir dermed raskt det endelige nettverket. Grensene som brukes for å finne hvilke nevroner som kan fjernes er spesifisert i prosent og dermed mindre problemavhengig å finne.

Det er gjennomført simuleringer med de to foreslåtte beskjæringsmetodene i en lærende invers kinematisk kontroller for styring av de tre første leddene til en ABB IRB140 manipulator. Resultatet var at å implementere beskjærte RBF nettverk i kontrolleren gjorde den mer robust mot systemets usikkerhet på grunn av økt generaliseringsevne. For de tilfellene med umodellert dynamikk ga de beskjærte nettverkene bedre banefølgning i forhold til bruk av feil system modell, ubeskjærte RBFNN'er og en type voksende nettverk kalt RANEKF. Beregningskostnadene ble også redusert da beskjæringsmetodene ble inkludert.

NTNU har et eksemplar av manipulatoren ABB IRB140 og den lærende inverse kinematiske kontrolleren med beskjæring av RBF nettverk bør implementeres og testes på den.

Contents

1	Introduction	1
1.1	Background Motivation	1
1.2	Purpose of This Report	2
1.3	Outline	3
1.4	Contributions	3
2	Background Theory and Notation	5
2.1	Introduction	5
2.2	Radial Basis Function Neural Networks (RBFNN)	5
2.3	Training	7
2.4	Optimal Approximation	8
2.5	Localization Principle	8
2.6	Networks for Manipulator Dynamics	9
2.7	Ill-Posed Problems	11
2.8	Resource Allocation Network Extended Kalman Filter (RANEKF)	11
2.9	Notation	13
3	Literature Review - Pruning and Regularization	15
3.1	Introduction	15
3.2	Weight and Output Based Pruning	15
3.3	Complexity Based Pruning	17
3.4	Sensitivity and Saliency Based Pruning	17
3.4.1	Optimal Brain Surgeon (OBS)	18
3.4.2	Weight Based Saliency	21
3.4.3	Sensitivity Measurement by Fourier Series	21
3.4.4	Local Sensitivity Analysis	22
3.4.5	Online Pruning for Robot Manipulator Tracking	23
3.5	Regularization	24

4	Weight Magnitude and Neuron Output Pruning	27
4.1	Introduction	27
4.2	Weight Magnitude Pruning	28
4.2.1	When to start pruning	28
4.2.2	Pruning	29
4.2.3	After Pruning	29
4.2.4	When to End Pruning	30
4.3	Neuron Output Pruning	30
4.3.1	When to start pruning	31
4.3.2	Pruning	31
4.3.3	After Pruning	31
4.3.4	When to End Pruning	32
4.4	Combined Weight Magnitude and Neuron Output Pruning	32
4.4.1	Both Pruning Criteria Met	32
4.4.2	Only One Pruning Criteria Met	32
4.4.3	Mixed Method	33
4.5	General Algorithm	33
5	Simulations with Pruning for Cross Function Learning	34
5.1	Introduction - Cross Function	34
5.2	RBF Network	35
5.2.1	Size	36
5.2.2	Training	39
5.2.3	Network Used Further	41
5.3	Results	41
5.3.1	Only Weight Magnitude Pruning	42
5.3.2	Only Neuron Output Pruning	45
5.3.3	Both Pruning Methods - Separately	47
5.3.4	Both Weight Magnitude Threshold and Neuron Output Pruning Threshold met	49
5.4	Discussion	50
5.4.1	Training, Size and Approximation Error	51
5.4.2	Pruning Methods	52
6	Online Learning Controller for a 2 DOF Manipulator	55
6.1	Introduction	55
6.2	2 Dof Robot Model	56
6.3	Controller and Desired Trajectory	56
6.4	Radial Basis Function Neural Network	59
6.5	Simulations with Correct Model and Unpruned RBFNNs	60
6.5.1	Correct Model	60

6.5.2	Unpruned RBFNN	62
6.6	Simulations with Online Pruning	62
6.6.1	Weight Magnitude Pruned RBFNN	64
6.6.2	Neuron Output Pruned RBFNN	65
6.6.3	Mixed Methods Pruned RBFNN	67
6.7	Simulations with Friction and a Constant Disturbance	68
6.7.1	Incorrect Model and Unpruned Networks	69
6.7.2	Online Pruning	70
6.8	Discussion	72
7	Online Learning Controller for ABB IRB140	76
7.1	Introduction	76
7.2	ABB IRB140	77
7.3	Controller and Desired Trajectory	77
7.4	RBF networks	79
7.5	Simulations with Correct Model and Unpruned RBFNN	80
7.5.1	Correct Model	81
7.5.2	Unpruned Radial Basis Function Neural Networks	82
7.6	Simulations with Online Pruning	83
7.6.1	Weight Magnitude Pruned RBFNN	84
7.6.2	Neuron Output Pruned RBFNN	86
7.6.3	Mixed Methods Pruned RBFNN	88
7.7	Simulations with Resource Allocation Network EKF	90
7.8	Simulations with Friction and a Constant Disturbance	93
7.8.1	Incorrect Model and Unpruned Networks	94
7.8.2	Online Pruning	95
7.8.3	Resource Allocation Networks EKF	98
7.9	Ill-Posed Learning and Weight Convergence	99
7.10	Discussion	103
7.10.1	Results	103
7.10.2	Simulation Time	107
7.10.3	Ill-Posed Problem	107
8	Concluding Remarks and Future Work	109
8.1	Conclusion	109
8.2	Further Work	110
	Bibliography	113

A	Stability Proves	114
A.1	Stability of Inverse Kinematic Controller	114
A.2	Stability of Learning Inverse Kinematic Controller	116
A.2.1	RBF Networks for Learning System Dynamics	116
A.2.2	Error Equation	118
A.2.3	Lyapunov Function Candidate	119
B	ABB IRB140 Data Sheet	122
C	Contents of Zip File	125

List of Figures

2.1	Hidden Layer	7
5.1	Cross Function	35
5.2	Too Small Networks	36
5.3	Too Small Network 2	37
5.4	Appropriate Size	38
5.5	Networks Trained too Little	39
5.6	Networks Trained too Little 2	40
5.7	Over Pruned by Weight Magnitude Pruning	44
5.8	Over Pruned by Node Output Pruning	47
5.9	Over Pruned by Both Weight and Node Output Pruning	50
6.1	2 DOF Revolute Joints Manipulator	57
6.2	Desired Trajectory End Effector	58
6.3	Desired Trajectory Joint 1 and Joint 2	58
6.4	Trajectory End Effector Using Correct Model	61
6.5	Trajectory Joint 1 and 2 Using Correct Model	61
6.6	Trajectory End Effector Using Unpruned RBFNNs	62
6.7	Trajectory Joint 1 and 2 Using Unpruned RBFNNs	63
6.8	Trajectory Joint 1 and 2 Using Weight Pruned RBFNNs	65
6.9	Trajectory Joint 1 and 2 Using Node Output Pruned RBFNNs	66
6.10	Trajectory Joint 1 and 2 Using Mixed Methods Pruned RBFNNs	68
6.11	Friction and Disturbance - Trajectory End Effector Using Incorrect Model	69
6.12	Friction and Disturbance - Trajectory Joint 1 and 2 Using Incorrect Model	70
6.13	Friction and Disturbance - Trajectory Joint 1 and 2 Using Weight Pruned RBFNNs	71
7.1	ABB IRB140	78
7.2	Desired Trajectory for ABB IRB140	79
7.3	Trajectory all 3 Joints Using Correct Model	82

7.4	Trajectory all 3 Joints Using Unpruned RBFNN	83
7.5	Trajectory all 3 Joints Using Weight Pruned RBFNN	87
7.6	Trajectory all 3 Joints Using Neuron Output Pruned RBFNN	89
7.7	Trajectory all 3 Joints Using Mixed Methods Pruned RBFNN	91
7.8	Trajectory all 3 Joints Using RANEKFs	93
7.9	Friction and Disturbance - Trajectory all 3 Joints Using Mixed Methods Pruning	95
7.10	Norm of Weights some \hat{M} Networks	101
7.11	Norm of Weights \hat{G} Networks	101
7.12	Norm of Weights Different \hat{c}_{23} Networks	102

List of Tables

5.1	Pruning Results for Weight Based Pruning	42
5.2	Pruning Results for Neuron Output Based Pruning	45
5.3	Pruning Results for Both Methods in Same Pruning	48
6.1	RBF Neural Networks	60
6.2	Unpruned Networks	63
6.3	Weight Magnitude Pruned Networks	64
6.4	Neuron Output Ratio Pruned Networks	66
6.5	Mixed Weight Magnitude and Output Ratio Pruned RBFNNs	67
6.6	Trajectory Tracking and Approximation Errors	72
6.7	Friction and Disturbance - Trajectory Tracking Errors	73
7.1	ABB IRB140 RBF Networks	81
7.2	Unpruned Networks	84
7.3	Weight Magnitude Pruned Networks	86
7.4	Neuron Output Pruned Networks	88
7.5	Mixed Methods Pruned Networks	90
7.6	Resource Allocation Networks EKF	92
7.7	Friction and Disturbance - Node Output Pruning	97
7.8	Friction and Disturbance - Mixed Method Pruning	98
7.9	Friction and Disturbance - RANEKFs	99
7.10	Trajectory Tracking and Approximation Errors 6π Simulation Time	104
7.11	Trajectory Tracking and Approximation Errors 20π Simulation Time	105
7.12	Disturbance and Friction - Trajectory Tracking and Approximation Errors	106

Chapter 1

Introduction

1.1 Background Motivation

Over the last years robot manipulators have become a very important part of the industry and are used all over the world in factories and places with high risk involved for humans. Thus also the subject of how to control the manipulators in a best possible way has become an active research field. The desired objective is to obtain fast and accurate manipulators that are safe and stable.

Implementing a model of the manipulator in the control scheme has shown to give better performance than non-model based controllers. [1] The precision of the model-based regulators is closely linked to the accuracy of the dynamic model and the parameters in it. Finding a completely correct model is however not an easy task due to the robot manipulator itself and also the surrounding environment are containing a great deal of uncertainties and disturbances.

Controllers utilizing *learning* include methods coping well with the uncertainties and modeling errors. These control systems have the ability to improve their future performance based on obtained knowledge from the past control tasks done by the manipulator. Out of the different learning schemes currently existing *Artificial Neural Networks* have proven to be well suited for storing the experienced knowledge of the system. They also have the ability to generalize from the knowledge on situations to new and unknown tasks similar to the already experienced ones.

Radial Basis Function Neural Network is a type of artificial neural networks that due certain properties like the known ability of universal approximation [24] is appropriate for use in robot manipulator control. They are also reported to be computationally more efficient than multilayer perceptron networks in a number of control applications [25].

The performance of the radial basis function neural network however is very

dependent on the size of the network and the main problem is the *Curse of Dimensionality* [10]. To give an accurate approximation the networks have to be large enough. On the other hand if they should become too large the problem of *over-fitting* is likely to occur. Over-fitting is when there are too many hidden neurons and the network starts to fit the errors within the training data in addition to the underlying function. This gives poor generalization ability for the network and in the worst cases the outputs become completely incorrect.

Another major problem following the curse of dimensionality is the computational cost of implementing radial basis function networks. When used in robot manipulator control it is absolutely necessary that the computations are fast enough so the outputs can be used in the regulator.

In general there are two ways to create a network with appropriate size, 1) *growing* and 2) *pruning*. The first starts with a small or empty neural network and adds neurons until some threshold for the approximation error is met. The second starts with an over dimensioned network and removes the weights and/or neurons that not are necessary for the network in order to be able to approximate. Pruning for artificial neural networks is based on the same concept as what naturally happens in the human brain when synaptic weights are pruned.

Artificial neural networks are created based on the neural network in human brains and the knowledge is stored in the synaptic weights for both the neural networks. For humans some synapses are cut to increase the brains ability to generalize from situation to situation when the humans are around the age of 20. The remaining weights in the brain then have their strength increased. If however the brains are pruned too much the sickness schizophrenia is developed which causes the brain to over-generalize and give outputs with no real meaning and little sense. [11]

1.2 Purpose of This Report

The purpose of this report is to create pruning methods for radial basis function neural networks that are suited for use in online learning control of robot manipulators. The system dynamic model of a manipulator is learned and then utilized in a model-based control scheme.

Focus in much of the pruning algorithms developed so far is on finding the optimal network with best possible generalization ability and smallest possible approximation error. They are however not concerned with the computational cost the pruning algorithm needs or how computational demanding the final network is.

For use in real-time control of a robot manipulator it is crucial that pruning happens fast since this will be done online while controlling the manipulator. In this setting it is probably better with a small network that perhaps not gives the best possible estimations but can give a satisfyingly approximation fast enough to be used in the controller.

1.3 Outline

First some theory on the radial basis function neural networks used in this report is given in Chapter 2 along with some other essential background information.

Then in Chapter 3 a literature review on pruning follows.

In Chapter 4 two pruning methods suited for use in online learning of the system dynamics of a manipulator are proposed. The first pruning method is *Weight Magnitude Pruning* which uses the magnitude of the weights to find the weights that not are necessary in the network and remove these. The second pruning method is *Neuron Output Pruning*. This method is based on the output of the activation function in the hidden neurons and finds which nodes that may be removed by looking at these outputs.

The two proposed pruning methods are tested in offline simulations for learning a cross function with a radial basis function network in Chapter 5.

Simulations with the two pruning methods for a 2 dof manipulator are done in Chapter 6. Here the manipulator is controlled with a learning inverse kinematic controller for a trajectory tracking task. The learning of the system dynamics consists of training and pruning the radial basis function networks online while tracking a desired trajectory.

In Chapter 7 the same learning inverse kinematic controller is simulated for control of the first 3 joints of the ABB IRB140 manipulator. Again the radial basis function neural networks are trained and pruned with the proposed pruning schemes online for learning the dynamics of the manipulator. Simulations with a growing neural network called *Resource Allocation Network Extended Kalman Filter* are also done. The obtained results from pruning and growing the networks then are compared.

Finally in Chapter 8 some conclusions are drawn and future work mentioned.

1.4 Contributions

- Two simple pruning methods for radial basis function neural networks in on-line learning control for a robot manipulator. Most of the removable neurons

are pruned during the first period. Thresholds are specified as a percent and thus less problem dependent to find. The proposed pruning methods are

1. *Weight Magnitude Pruning* which is based on a method in [8]. It has however been enhanced with *a)* finding the threshold as a percent of the L^2 norm of the weights, and *b)* using a growing threshold so additional neurons are removed after the first pruning.
 2. *Neuron Output Pruning* which in a new setting uses neuron output ratio from [2]. Here the neuron output ratios are summed together for each unit and a pruning threshold is found as a percent of the maximum output ratio sum.
- Simulations in Matlab/Simulink that demonstrate the efficiency of using the proposed pruning methods in a learning inverse kinematic controller.

Chapter 2

Background Theory and Notation

2.1 Introduction

The neural networks in this report are *Radial Basis Function Neural Networks* and they are often referred to as either RBF networks or RBFNNs. A very brief description of the RBF networks used in this report will in this chapter be given. Some general properties of the radial basis function networks are also mentioned as they are important later in the report. More on Radial Basis Function Neural Networks and other neural networks can be found in [10].

How the dynamics of a manipulator are learned by RBF networks is described in section 2.6. Then a short definition of an ill-posed problem follows in the next section.

In order to compare the pruned RBF networks a growing network called *Resource Allocation Network Extended Kalman Filter (RANEKF)* is used later in this report. Thus some short background theory on RANEKF will be given in this chapter.

At the end a small list of different notations used trough the report can be found.

2.2 Radial Basis Function Neural Networks (RBFNN)

RBF networks are feed-forward neural networks with one input layer, one output layer and only one hidden layer. [10] The output of the an output neuron is given as

$$F_j(x) = \sum_{i=1}^N w_{ji} a_i(\|x - \mu_i\|) \quad (2.2.1)$$

where there are N units in the hidden layer, w_{ji} is the weight between output neuron j and hidden unit i , $a_i(\|x - \mu_i\|)$ is the activation function in hidden unit i to the input x and μ_i is the center of node i . If there only is one output neuron in the network the subscript j can be removed and the final network output may be written as

$$F(x) = \sum_{i=1}^N w_i a_i(\|x - \mu_i\|) \quad (2.2.2)$$

All the networks only have one output node in this report and this gives one single weight connecting a hidden node with the output layer. This again implies that there are the same number of hidden neurons and weights between hidden and output layer. These weights are the only ones that are changed while the weights between the input layer and the hidden layer are fixed at one and not given any attention here.

The changeable weights (from now on only referred to as the weights) can be written in a vector as

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} \quad (2.2.3)$$

where w_1 is the weight between hidden unit 1 and the output unit, w_2 is the weight between hidden unit 2 and the output unit and N is the number of units in the hidden layer. All the weights are initialized to zero in the beginning.

The activation function used is the *gaussian* function and the activation function for node i is given as

$$a_i(\|x - \mu_i\|) = k_g e^{-\frac{\|x - \mu_i\|^2}{\sigma^2}} \quad (2.2.4)$$

where k_g is a constant, x is the input to the network, μ_i is the centre of node i , $\|\cdot\|$ is the 2-norm or euclidean distance and σ is the width (also referred to as spread) of the activation function. Here the width the whole time is fixed and taken as the distance to the next unit. These activation functions can also be written in a vector

$$\mathbf{a}(\|x - \mu\|) = \begin{bmatrix} a_1(\|x - \mu_1\|) \\ a_2(\|x - \mu_2\|) \\ \vdots \\ a_N(\|x - \mu_N\|) \end{bmatrix} \quad (2.2.5)$$

and the centres of the hidden neurons can be written as

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_N \end{bmatrix} \quad (2.2.6)$$

To simplify the writing $\mathbf{a}(\|x - \mu\|)$ will now be written as $\mathbf{a}(x)$ since the centres are fixed.

The centres are placed on a regular lattice with fixed place as shown in figure 2.1 below. Two inputs give a 2-dimensional lattice for the hidden neurons as the figure has.

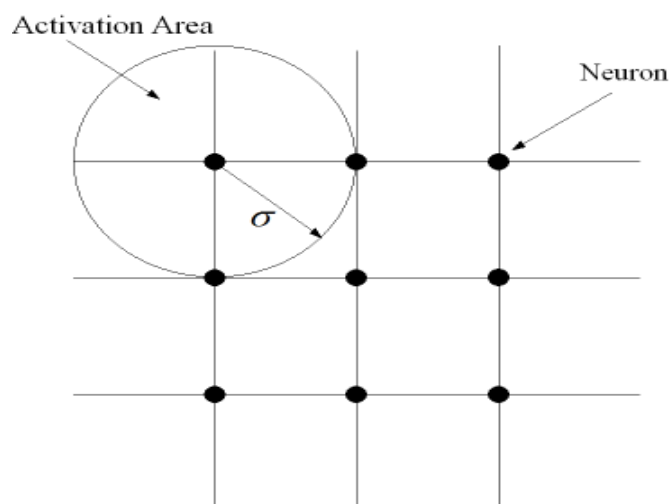


Figure 2.1: Hidden Layer

2.3 Training

The networks are trained with supervised learning using a *gradient descent* algorithm in this report. As already mentioned the only weights that are changed are those between the hidden layer and the output layer.

The new weights are found as this

$$\mathbf{w} = \mathbf{w} + \alpha \Delta \mathbf{w} \quad (2.3.1)$$

where \mathbf{w} are the current weights, α is a training rate and $\Delta\mathbf{w}$ is the update of the weights. This update is found minimizing the cost function

$$K = \frac{1}{2}(z - \hat{z})^2 = \frac{1}{2}e^2 \quad (2.3.2)$$

where z is the desired output of the network while \hat{z} is the actual output and $e = z - \hat{z}$ is the approximation error. The minimization gives

$$\Delta\mathbf{w} = -e\mathbf{a}(x) \quad (2.3.3)$$

where $\mathbf{a}(x)$ is the activation function output for the input x .

2.4 Optimal Approximation

A radial basis function neural network is known to be able to approximate any function with any given accuracy as long as the network is large enough and has the correct chosen properties, like for instance the width of the activation function. See [24] for a proof of that RBF networks with one hidden layer are capable of universal approximation.

The optimal network output can be written as

$$F(x) = \mathbf{w}^{*T}\mathbf{a}(x) + \varepsilon^* \quad (2.4.1)$$

where \mathbf{w}^* are the optimal weights and ε^* is the optimal estimation error which can be made arbitrary small.

2.5 Localization Principle

One of the main differences between Multilayer Perceptron (MLP) networks and RBF networks is that the activation functions in RBF networks are *radial basis functions (RBF)*. These functions are a special set of real-valued functions which value depends only on the distance from an input to another point called a center. This distance is usually taken as the Euclidean distance. More on radial basis functions can be found in [23].

In equation 2.2.4 which is the gaussian activation function used for the networks in this report the width, σ , specifies the area for where a neuron fires. The

area for where a neuron fires or is activated is called an activation area. See figure 2.1. So for an input will only those neurons closer than σ contribute to the total output of the network while nodes further away have little or none influence for that specific input. This is called the *Localization Principle* since which units that fire are based on their localization.

2.6 Networks for Manipulator Dynamics

The dynamic equation for a robot manipulator can be written as this

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) = \tau \quad (2.6.1)$$

where q, \dot{q}, \ddot{q} are the joint position, velocity and acceleration respectively. $M(q)$ is the inertia matrix, $C(q, \dot{q})$ is a matrix containing the centrifugal and coriolis forces, $G(q)$ is the gravitation matrix and τ is the control torque. This equation can also be written as

$$\begin{bmatrix} m_{11} & \cdots & m_{1n} \\ \vdots & \ddots & \vdots \\ m_{n1} & \cdots & m_{nn} \end{bmatrix} \begin{bmatrix} \ddot{q}_1 \\ \vdots \\ \ddot{q}_n \end{bmatrix} + \begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \vdots \\ \dot{q}_n \end{bmatrix} + \begin{bmatrix} g_1 \\ \vdots \\ g_n \end{bmatrix} = \begin{bmatrix} \tau_1 \\ \vdots \\ \tau_n \end{bmatrix} \quad (2.6.2)$$

where n is the number of joints for the manipulator. Due to all the joints being activated in this report the number of degrees of freedom is the same as the number of joints.

The elements in the dynamic matrices are separately learned by a RBF network in this report, that is one network is dedicated to one element in one of the matrices. Then there initially will be $n \times n + n \times n + n$ RBF networks. When the manipulator has many degrees of freedom (dof) it follows that there are very many networks. However due to the symmetric property of the inertia matrix, $M = M^T$, not all the elements need to be approximated by a RBFNN.

In order to simplify the writing for the RBFNN estimated matrices some notations based on definitions in [7] is used, and following are some explanations.

$$\hat{M} = W_M^T \bullet A_M(x) \quad (2.6.3)$$

where W_M is the weight matrix defined as

$$W_M^T \triangleq \begin{bmatrix} \mathbf{w}_{m11}^T & \cdots & \mathbf{w}_{m1n}^T \\ \vdots & \ddots & \vdots \\ \mathbf{w}_{mn1}^T & \cdots & \mathbf{w}_{mnn}^T \end{bmatrix} \quad (2.6.4)$$

when the elements in the matrix, such as \mathbf{w}_{m11} , are a column vectors as defined in equation (2.2.3). $A_M(x)$ is the activation function matrix for an input x , defined as

$$A_M(x) \triangleq \begin{bmatrix} \mathbf{a}_{m11}(x) & \cdots & \mathbf{a}_{m1n}(x) \\ \vdots & \ddots & \vdots \\ \mathbf{a}_{mnn}(x) & \cdots & \mathbf{a}_{mnn}(x) \end{bmatrix} \quad (2.6.5)$$

where the elements, like \mathbf{a}_{m11} , are as in equation (2.2.5).

When calculating \hat{M} as in equation (6.4.1) the \bullet between W_M and $A_M(x)$ operator shows that this is not a normal matrix multiplication but used as this

$$W_M^T \bullet A_M(x) = \begin{bmatrix} \mathbf{w}_{m11}^T \mathbf{a}_{m11}(x) & \cdots & \mathbf{w}_{m1n}^T \mathbf{a}_{m1n}(x) \\ \vdots & \ddots & \vdots \\ \mathbf{w}_{mnn}^T \mathbf{a}_{mnn}(x) & \cdots & \mathbf{w}_{mnn}^T \mathbf{a}_{mnn}(x) \end{bmatrix} \quad (2.6.6)$$

So for the network that approximates the element m_{11} the output is given as $\hat{m}_{11} = \mathbf{w}_{m11}^T \mathbf{a}_{m11}(x)$.

Finding the approximated C matrix follows the same pattern

$$\hat{C} = W_C^T \bullet A_C(x) \quad (2.6.7)$$

$$\Rightarrow \begin{bmatrix} \hat{c}_{11} & \cdots & \hat{c}_{1n} \\ \vdots & \ddots & \vdots \\ \hat{c}_{n1} & \cdots & \hat{c}_{nn} \end{bmatrix} = \begin{bmatrix} \mathbf{w}_{c11}^T & \cdots & \mathbf{w}_{c1n}^T \\ \vdots & \ddots & \vdots \\ \mathbf{w}_{cn1}^T & \cdots & \mathbf{w}_{cnn}^T \end{bmatrix} \bullet \begin{bmatrix} \mathbf{a}_{c11}(x) & \cdots & \mathbf{a}_{c1n}(x) \\ \vdots & \ddots & \vdots \\ \mathbf{a}_{cn1}(x) & \cdots & \mathbf{a}_{cnn}(x) \end{bmatrix} \quad (2.6.8)$$

$$= \begin{bmatrix} \mathbf{w}_{c11}^T \mathbf{a}_{c11}(x) & \cdots & \mathbf{w}_{c1n}^T \mathbf{a}_{c1n}(x) \\ \vdots & \ddots & \vdots \\ \mathbf{w}_{cn1}^T \mathbf{a}_{cn1}(x) & \cdots & \mathbf{w}_{cnn}^T \mathbf{a}_{cnn}(x) \end{bmatrix} \quad (2.6.9)$$

and so does also the estimation of G

$$\hat{G} = W_G^T \bullet A_G(x) \quad (2.6.10)$$

$$\Rightarrow \begin{bmatrix} \hat{g}_1 \\ \vdots \\ \hat{g}_n \end{bmatrix} = \begin{bmatrix} \mathbf{w}_{g1}^T \\ \vdots \\ \mathbf{w}_{gn}^T \end{bmatrix} \bullet \begin{bmatrix} \mathbf{a}_{g1}(x) \\ \vdots \\ \mathbf{a}_{gn}(x) \end{bmatrix} = \begin{bmatrix} \mathbf{w}_{g1}^T \mathbf{a}_{g1}(x) \\ \vdots \\ \mathbf{w}_{gn}^T \mathbf{a}_{gn}(x) \end{bmatrix} \quad (2.6.11)$$

2.7 Ill-Posed Problems

An *ill-posed* problem is basically when the training data contains too little information about the desired solution to give enough variation in the training of the networks. In [10] Haykin identifies three conditions which have to be satisfied in order for a problem to be well-posed and thus not ill-posed. First it is assumed that there are two metric spaces, a domain X and a range Y , and a fixed but unknown function f mapping between them. The three conditions then are:

1. *Existence.* For every input vector $\mathbf{x} \in X$ there exist an output $y = f(x)$, where $y \in Y$.
2. *Uniqueness.* For any pair of input vectors $\mathbf{x}, \mathbf{t} \in X$ then $f(\mathbf{x}) = f(\mathbf{t})$ iff $\mathbf{x} = \mathbf{t}$.
3. *Continuity.* For any $\epsilon > 0$ there exists a $\delta = \delta(\epsilon)$ such that $\rho_x(\mathbf{x}, \mathbf{t}) < \delta$ implies that $\rho_y(f(\mathbf{x}), f(\mathbf{t})) < \epsilon$, where $\rho(\cdot, \cdot)$ is the distance between the symbols and their respective spaces. This property is also referred to as *stability*.

2.8 Resource Allocation Network Extended Kalman Filter (RANEKF)

The other way to handle the uncertainties of finding a network with appropriate size and placement for the neurons is a self-growing network. One of the growing RBF networks that exist is *Resource Allocation Network (RAN)* which in an article by Platt [26] was proposed. This network later was enhanced to use an *Extended Kalman Filter (EKF)* algorithm for the adaptation stage in [15] and here a brief description of the Resource Allocation Network Extended Kalman Filter (RANEKF) will be given. RANEKF is in simulations later in this report implemented in a learning controller and the result from this is compared to the result obtained with the pruned networks. The information on RANEKF here is found in [6] and [15].

A RANEKF starts with no neurons in the network and adds one neuron at the time during a training period. The adding of a new neuron is based the localization to the inputs and approximation error for the network. This gives that the learning process requires allocation of a new hidden node as well as adaptation of the other network parameters as weights and width of the activation function. Thus the learning scheme can be viewed as a hybrid learning scheme

consisting of an unsupervised learning part for choosing the placement of the new neurons and a supervised learning scheme for determining the network parameters.

The output of a RANEKF is given as

$$f(x) = \alpha_0 + \sum_{k=1}^K \alpha_k \phi_k(x) \quad (2.8.1)$$

where $x \in R^m$ is the input and K is the number of hidden units. α_k is the connecting weight from k th hidden unit to the output layer, and α_0 is a bias term. The activation function is taken as the gaussian function given as

$$\phi_k(x) = e^{-\frac{1}{2\sigma^2} \|x - \mu_k\|^2} \quad (2.8.2)$$

where μ_k is the center vector for the k th hidden unit and σ is the width of the gaussian function.

The criteria used for deciding whether a new unit should be added to the hidden layer are the following three:

$$\|x_n - \mu_{nr}\| > \epsilon_n \quad (2.8.3)$$

$$e_n = y(n) - \hat{y}(n) > e_{min} \quad (2.8.4)$$

$$e_{rmsn} = \sqrt{\frac{\sum_{i=n-(M-1)}^n [y(i) - \hat{y}(i)]^2}{M}} > e'_{min} \quad (2.8.5)$$

where μ_{nr} is the center of the hidden neuron with the smallest distance to the input x_n . M is a sliding window used for calculating the RMS output approximation error, e_{rmsn} . ϵ_n , e_{min} and e'_{min} are threshold values that are set to appropriate values in order to find out if a new neuron is added or not. All the three criteria have to be met at for the same input if the number of hidden units is to be increased. Said in another way represents the first criterion that the shortest distance from a new input to a node has to larger than ϵ_n if a new hidden unit should be added. The second says that the approximation error for the network to that given input has to be larger than e_{min} , and the third criterion gives that the rms approximation error over the M last inputs has to be larger than e'_{min} .

A new input is assigned the following parameter value when added

$$\alpha_{K+1} = e_n \quad (2.8.6)$$

$$\mu_{K+1} = x_n \quad (2.8.7)$$

$$\sigma_{K+1} = \kappa \|x_n - \mu_{nr}\| \quad (2.8.8)$$

where κ is an overlap factor that decides the overlap of the response of a hidden unit in the input space.

If the criteria for adding a new neuron are not met the algorithm uses an extended Kalman filter (EKF) to adjust the parameters in the network, $w = [\alpha_0, \alpha_1, \mu_1^T, \sigma_1, \dots, \alpha_K, \mu_K^T, \sigma_K]^T$. The update law is given as

$$w_n = w_{n-1} + k_n e_n \quad (2.8.9)$$

where k_n is a kalman filter gain vector decided by

$$k_n = P_{n-1} a_n [R_n + a_n^T P_{n-1} a_n]^{-1} \quad (2.8.10)$$

where a_n is a gradient vector on the form

$$a_n = \nabla_w f(x_n) \quad (2.8.11)$$

$$= [1, \phi_1(x_n), \phi_1(x_n) \frac{2\alpha_1}{\sigma_1^2} (x_n - \mu_1)^T, \quad (2.8.12)$$

$$\phi_1(x_n) \frac{2\alpha_1}{\sigma_1^3} \|x_n - \mu_1\|^2, \quad (2.8.13)$$

$$\phi_K(x_n), \phi_K(x_n) \frac{2\alpha_K}{\sigma_K^2} (x_n - \mu_K)^T, \quad (2.8.14)$$

$$\phi_K(x_n) \frac{2\alpha_K}{\sigma_K^3} \|x_n - \mu_K\|^2]^T, \quad (2.8.15)$$

R_n is the variance of the measurement noise and P_n is the error covariance matrix updated as

$$P_n = [I - K_n a_n^T] P_{n-1} + QI \quad (2.8.16)$$

where Q is a scalar that determines the step size in the gradient direction.

2.9 Notation

- RBF - Radial Basis Function

- NN - (Artificial) Neural Network
- MLP - Multilayer Perceptron, neural network capable of universal approximation with an input layer, an output layer and one or more hidden layers
- Neuron, Node, Center and Unit refers to the same
- Size of a RBF network is the number of neurons in the hidden layer
- Norm - the norm of the weights is taken as the L^2 norm
- One training iteration is taking one training sample, feeding the input to the network, comparing the output with the desired output and then change the weights according to the estimation error
- One period is first to prune the network once and then train the network with one training set
- RMSE - Root Mean Square Error, used to measure the approximation errors
- SSE - Sum Square Error, used to calculate the tracking error
- DOF - Degree Of Freedom, for a manipulator
- RANEKF - Resource Allocation Network Extended Kalman Filter

Chapter 3

Literature Review - Pruning and Regularization

3.1 Introduction

A review of some currently existing pruning schemes will in this chapter be given. The goal with pruning often is to find the point of where the network approximation error is smallest possible and the generalization ability highest possible. Hence the focus in pruning have been on finding the most optimal network. These pruning methods often are very time consuming and the final network can be quite large.

Pruning methods start with an initially large network and train this to a minima or close to a minima. Then based on different criteria they find the network parameter which can be removed. By network parameter either a weight or a neuron is here meant. Some of the pruning methods remove weights only while other remove a neuron and thus also its belonging weights.

Many of the neural networks are not radial basis function networks in this chapter and the networks may then have changeable weights between several layers. In general the networks are feed-forward neural networks.

3.2 Weight and Output Based Pruning

The output of an entire network depends on the output of the neurons and the size of the weights. Thus to look at those two factors when deciding which network parameters that not are necessary for the network will make sense.

Some of the earliest pruning versions are based on the magnitude of the weights, like in an article by Hagiwara [8]. This article describes three different pruning methods that are simple in their logic and also not very computational demanding.

The first is the *Goodness Factor Method* which is based on that an important neuron excites frequently and has strong weights connecting to the next layer. Pruning is done with a Goodness Factor defined as $G_i^k = \sum_p \sum_j (w_{ji}^k o_i^k)^2$ for neuron i in layer k . w_{ji}^k is the weight between the i th neuron in the k th layer and the j neuron in the $(k + 1)$ th layer, while o_i^k is the output of the i th neuron in the k th layer. The neuron with the lowest goodness factor is after training removed, the network then retrained and another neuron with the currently lowest goodness factor can be found. This is repeated for some undefined amount of time.

Based on the first a second method called *Consuming Energy Method* is proposed. The difference here is that the consuming energy method includes the thought that an important neuron not only excites frequently itself. Neurons in the following layer will also be excited. The consumed energy for the i th neuron in the k th layer is defined as $E_i^k = \sum_p \sum_j w_{ji}^k o_j^{k+1} o_i^k$, where w_{ji}^k is the weight between the i th neuron in the k th layer and the j neuron in the $(k + 1)$ th layer, o_i^k is the output of the i th neuron in the k th layer and o_j^{k+1} is the output of the j th neuron in layer $k + 1$.

The third pruning method described in the article by Hagiwara is called *Weights Power Method* and is a simplified version of the two pruning methods already mentioned. Here only the weights are considered and the weight power for neuron i in layer k is defined as $W_i^k = \sum_j (w_{ji}^k)^2$.

All three versions are simulated for the same problem and then compared. The result was that the weights power method gave the smallest network and also the best generalization result for both neuron pruning and weight pruning.

Later it has been argued that these methods are too simple. There is no guarantee for the right weight or neuron being removed due to a neuron with small weights may be important for the overall performance and accuracy of the network. That is why the trend in the later years has moved towards pruning based on some kind of complexity measurement of the network or sensitivity analysis of the different network parameters.

3.3 Complexity Based Pruning

Complexity of a brain or a neural network is a measurement of how well neurons have been able to group together to form clusters for interpreting different inputs and how well these groups are able to function. One example of an algorithm that prunes based on the complexity can in an article by Zhang and Qiao [30] be found. This algorithm finds the the least important neuron by first removing one node, calculating the change in neural complexity, placing this neuron back into the network for so to do the same with another unit. When all of the neurons have been removed and restored the neuron with the highest complexity and hence the least important one is found and removed. This is done until the minimum squared error is less than some objective error.

The entropy of the network is used to calculate the network complexity. This entropy is found from the covariance matrix of the neural network connection matrix and gives a measurement of how correlated the different neurons in the network are. If all neurons are either totally independent or dependent the complexity is zero and thus no neurons are pruned.

The algorithm described here is for a three layer feed-forward neural network which is a network with input layer, output layer and one hidden layer as RBF networks. Weights that not are unity from the input layer to the hidden layer are however here included. Focus have been more on MLPs than RBFs and the complexity pruning scheme is supposed to be easy to extend to larger networks.

3.4 Sensitivity and Saliency Based Pruning

Sensitivity analysis techniques are probably the most common group of pruning methods and cover a wide range of different sensitivity measurements. The concept for the sensitivity analysis methods is however the same. Find and remove the network parameters with the lowest sensitivity and thus have the least relevance for the final network output. To the sensitivity analysis the network parameter to be pruned is taken as the input and the output is usually the output of the entire network. Whether the network parameter used is an input neuron, hidden neuron or a single weight varies from method to method.

Saliency can be seen as kind of sensitivity measurement and is widely used in the field of pruning over the last years. Something that stands out compared to its surroundings is the meaning of the word saliency. How this measurement is

defined depends on the method it is implemented in.

Originally the idea to sensitivity based pruning was to define some sensitivity measurement for the whole network and calculate this when removing and reinserting one neuron at the time. How the sensitivity varied when removing the different neurons gave an idea of how sensitive the network was to losing that specific neuron.

A trimming method for networks called *skeletonization* was probably the first version and proposed in an article by Mozer and Smolensky [21]. Here the contribution of each weight is found as a sensitivity measurement and the weight with the least contribution is removed. The sensitivity for a weight, w_{ji} , from node i in one layer to node j in the following layer is found as

$$s_{ji} = J(\text{without } w_{ji}) - J(\text{with } w_{ji}) \quad (3.4.1)$$

where $J(\cdot)$ is a performance measurement.

Sensitivity and complexity based pruning are closely related to each other. The difference mainly is that the sensitivity analysis finds an estimate (or sometimes called saliency) for the sensitivity of each neuron to the network while complexity pruning sees how the complexity of the entire network changes when a neuron is removed. By using an estimate for the sensitivity there is not necessary to remove one neuron at the time and calculate the sensitivity for each, which has reduced the computational cost greatly.

3.4.1 Optimal Brain Surgeon (OBS)

The perhaps most used sensitivity pruning method and also has shown good results over time is called *Optimal Brain Surgeon (OBS)* and was proposed by Hassibi, Stork and Wolff [9]. OBS is based on a pruning algorithm called *Optimal Brain Damage (OBD)* by Le Cun, Denker and Solla [4] which proposed that the weight with the smallest saliency will generate the smallest error variation to remove.

Both Optimal Brain Damage and Optimal Brain Surgeon start with a network already trained to converge to a local minima and tries to minimize the cost function

$$E = \frac{1}{2} \sum_{j=1}^p \sum_{i=1}^M (d_i(j) - y_i(j))^2 \quad (3.4.2)$$

where $d_i(j)$ is the desired output, $y_i(j)$ is the actual output of output neuron i to

training sample j , p is the number of training samples and M is the number of output neurons.

Then the functional Taylor series of the error with respect to weights is found. This can be written as

$$\Delta E = \left(\frac{\partial E}{\partial w} \right)^T \Delta w + \frac{1}{2} \Delta w^T H \Delta w + O(\| \Delta w \|^3) \quad (3.4.3)$$

where H is the Hessian defined as

$$H \triangleq \frac{\partial^2 E}{\partial w^2} \quad (3.4.4)$$

Since the training has converged the first term in the Taylor series vanish. The third and all higher order term are also neglected. So the goal is to minimize the increase in error, ΔE as above, by setting one of the weights to zero. So far is both OBD and OBS doing the same. In OBD however an assumption that the hessian is diagonally dominant was made and thus only the diagonal elements could be chosen. The *saliency coefficient* was then found as the increase in error E when changing the weights by Δw

$$S_i = \Delta E = \frac{1}{2} \sum_i h_{ii} [\Delta w_{ii}]^2 \quad (3.4.5)$$

Hassibi et al found that the hessian in fact was far from diagonally dominant, something that resulted in OBD for some cases removed the wrong weights. Hence the assumption is not made in OBS and this is the main difference between those two methods. The saliency in OBS is found by solving the minimization problem with a Lagrangian multiplier and the saliency of weight q then becomes

$$L_q = \frac{1}{2} \frac{w_q^2}{[H^{-1}]_{qq}} \quad (3.4.6)$$

From the same minimizing problem the optimal weight change is also found which is used to update all the other weights that are not removed. This optimal weight change is given as

$$\Delta w = - \frac{w_q}{[H^{-1}]_{qq}} H^{-1} e_q \quad (3.4.7)$$

The weight with the lowest saliency is then pruned and the remaining weights are updated according to (3.4.7). Again the inverse hessian is calculated to find the weight with the smallest saliency and the optimal weight change. This is repeated to remove one weight at the time until there are no more weights that can be removed without causing a large increase in E .

From simulations done in [9] it was shown that OBS gave significantly better results than OBD and also weight based pruning.

Other Versions of Optimal Brain Surgeon

The problem with Optimal Brain Surgeon is the fact that the method is very time consuming due to the repeatedly calculation of the inverse hessian matrix. That is why it more recently has been created new variants of OBS and some is shortly mentioned here.

Stahlberger and Riedmiller have proposed a variant called *Generalized Optimal Brain Surgeon (G-OBS)* [29] that prunes a subset of m weights in one step instead of only one as earlier done. However they also found that if $m > 3$ the original OBS actually would be faster. So G-OBS could remove up to three weights in a single step and would then be a bit faster than OBS.

In the same article a special case of the G-OBS method called *Unit Optimal Brain Surgeon (U-OBS)* was also proposed. If possible this method chose the subset of weights to be removed as the weights belonging to the same neuron and could thus remove an entire unit in a single step. This could be done in a much faster time than the original OBS could remove a neuron.

In a paper by Attik [3] another version of OBS called *Flexible Optimal Brain Surgeon (F-OBS)* specialized in removing connections between input and hidden layer is proposed. The benefits with F-OBS are that the algorithm can reduce the number of weights between the variables and the hidden layer and also reduce the number of variables. Attik also combines this version with generalized OBS into a method called *General Flexible Optimal Brain Surgeon (GF-OBS)* which in one stage removes a subset of weights between the input and hidden layer. The results from F-OBS and GF-OBS compared to G-OBS and U-OBS were approximately equally well. Which pruning scheme to use would be depending on the problem and network to prune.

To speed up the OBS pruning Meng proposes an algorithm where the OBS pruning case is a penalty item of the network cost function [19]. The proposed scheme is called penalty OBS and has a lot in common with regularization.

In [31] by Zhao, Liu and Zang a version that uses pseudo-entropy of the weights as a penalty term during training to control the weights distribution to be uniformly distributed is described. The authors propose that with weights being uniformly distributed there are no need for adjusting remaining weights or re-training during and after pruning. Hence this combination of regularization and pruning gives a pruned network quicker then the original OBS. By simulation they also show an improvement of the generalization ability.

3.4.2 Weight Based Saliency

In an article by Messer and Kittler [20] they develop an algorithm called *Fast Unit Selection Algorithm (USA)* where a saliency is defined more as the importance of some network parameter than sensitivity of the parameter. The saliency of unit i is taken as the average square of the weight connecting that unit in the layer l to the units in the next layer, $(l + 1)$. This is given as

$$S_i = \frac{\sum_{j=1}^{n^{l+1}} (w_{ij})^2}{n^{l+1}} \quad (3.4.8)$$

where n^{l+1} is the number of units in layer $l + 1$. A higher saliency means the parameter is more important to the network. Both neurons and weights can be used for pruning and only one parameter is removed at each step.

The algorithm starts with two large enough identical networks, one pruning set and one for validation. Both sets are trained to an acceptable level of performance before the saliency of all neurons/weights for the validation set is calculated. The parameter with the lowest saliency is removed in the validation network and this network is then retrained with a small number of epochs. After this the errors for the original and for the validation set are found. If the error is smaller for the validation set the same neuron is removed for the pruning network and this network also retrained. Then the same procedure with finding a new parameter to prune is repeated. The stopping criterion is reached when removing a parameter for the validation network gives a higher error than for the pruning network.

3.4.3 Sensitivity Measurement by Fourier Series

Another sensitivity based pruning algorithm is proposed in a paper by Honggui and Junfei [12]. This method finds the relevance for each hidden node based on the *Extended Fourier Amplitude Sensitivity Test (EFAST)*. It is pruned for weights between the hidden layer and output layer. The algorithm is developed for feed-forward neural networks and aims at making them self-organizing. Pruning begins when the training has done some iterations instead of waiting until the training is complete.

The output of the network can be expressed as a polynomial expansion, which can be expanded into a Fourier series. In this series the input factor to the sensitivity test will correspond to the Fourier amplitude at the fundamental frequency. By using this Fourier decomposition the variance of the output can be found based on the input factor and Fourier coefficients. Further this variance is used to find

the percentage of sensitivity a neuron has of the total sensitivity in the network, and this percentage is called the *contribution ratio* for a neuron. All neurons with contribution ratio less than some threshold are pruned and thus may several neurons be pruned at once.

3.4.4 Local Sensitivity Analysis

Sensitivity pruning can be separated between *Local Sensitivity* and *Global Sensitivity*. The global methods look at the whole network to find the network parameter with the lowest sensitivity compared to all the network parameters of the same type in the network. Local sensitivity on the other hand compares only the neurons in the same layer or weights connecting the same layers. This is most useful for networks with many hidden layers.

The sensitivity pruning schemes mentioned so far have been global methods and a local version can be found in an article by Ponnappelli, Ho and Thomson [27]. Here the concept of *Local Relative Sensitivity Index (LRSI)* for weights was introduced, given as

$$LRSI_{ji} = \frac{|s_{ji}|}{\sum_{m=1}^M |s_{jm}|} \quad (3.4.9)$$

where i is a node in layer k , j is a node in layer $k + 1$, M is the total number of connections to node j from layer k and s_{ji} is the sensitivity estimate for the weight connecting node i in the k th layer and node j in the $(k + 1)$ th layer. The sensitivity estimate was taken as

$$s_{ji} = \sum_{t=0}^{T-1} [\Delta w_{ji}(t)]^2 \frac{w_{ji}^f}{\eta(w_{ji}^f - w_{ji}^i)} \quad (3.4.10)$$

where w_{ji}^f is the final value of weight w_{ji} when completely trained, T is the total number of training iterations, η is the training rate and Δw_{ji} is the change in weight w_{ji} for one training iteration. The computational cost for this estimate compared to OBS is relatively small since it only uses information that is available during backpropagation and not calculate the hessian.

LRSI is the ratio between the sensitivity of a particular weight and the sum of all the weights from the previously layer to the same node. It is looked at each node, and any weight with a LRSI smaller than some threshold value is pruned. After a pruning phase is a new training phase, and then the performance of the new network is compared against the performance of the network before pruning. If there is an improvement the pruning and retraining phases are repeated

until the sum squared error is small enough or the performance no longer improves.

In [28] *Local Parameter Variance Nullity (LPVN)* is proposed based on average local sensitivity over all possible patterns for either weights or nodes. If this variance of sensitivity for a given parameter is small the parameter will have little or no influence on the output of the network and can thus be pruned. The sensitivity is found as in (3.4.10).

Pruning starts after some training and removes all the parameters with LPVN smaller than some threshold value. The network is then trained more and the LVPN is again calculated. Before any other parameters are removed the pruning is evaluated by checking if the newly pruned network has better performance than the old network. If not the old network is restored and the same layer is pruned based on a reduced pruning list. If the pruned network shows a better performance the algorithm moves to the next hidden layer and the same procedure is done again. The pruning is over when there are no more hidden layers to prune.

The performance check is done by *Cross Validation*, a method proposed by Huynh and Setiono [14] and originally based on the magnitude of the weights. The whole data set is split into two parts, one training set, TR , and one validation set, CV . Both sets are pruned and the performance is found with for instance the root-mean square function for each set separately. This is denoted J'_{TR} for the pruned training set and J'_{CV} for the pruned validation set. The performance for both sets were also found before pruning, denoted as J_{TR} for the training set and J_{CV} for the validation set. A comparison is done as

$$\rho(\zeta J'_{TR} + J'_{CV}) < (\zeta J_{TR} + J_{CV}) \quad (3.4.11)$$

where ρ is a constant that gives priority to the pruned network and $\zeta < 1$ is another constant that encourages generalization ability. Utilizing an extra cross validation set is to ensure that pruning not only reduce the size of the network but also improve the generalization ability.

3.4.5 Online Pruning for Robot Manipulator Tracking

One pruning scheme for online training and pruning for a learning robot manipulator tracking controller can be found in an article by Ni, Song and Grimbale [22].

They propose a robust backpropagation training algorithm for online pruning based on the Optimal Brain Surgeon method (OBS). Saliency of a weight is found by considering a perturbation to that weight and then see what influence on the estimation error it gives if this weight is set to zero. Due to the saliency being defined as in OBS the inverse of the Hessian matrix is estimated each iteration.

OBS is in this article enhanced into being a unit pruning method instead of only pruning weights. This is done by checking all the outgoing weights from a node and if all of these can be pruned then the whole unit may be removed.

The manipulator is described in a discrete-time system given as

$$x_1(k+1) = x_2(k) \quad (3.4.12)$$

$$\vdots \quad (3.4.13)$$

$$x_{n-1}(k+1) = x_n \quad (3.4.14)$$

$$x_n(k+1) = f(x(k)) + u(k) + d(k) \quad (3.4.15)$$

where $f(x(k)) \in R^m$ is a nonlinear function, $u(k) \in R^m$ is the control signal vector and $d(k) \in R^m$ is a bounded disturbance vector.

To train the network an estimation error is used and this is found by feeding the tracking error into a filter. The tracking error is defined as

$$e_n(k) = x_n(k) - x_{nd}(k) \quad (3.4.16)$$

where $x_{nd}(k)$ is the desired trajectory and $x_n(k)$ is the actual trajectory. The an estimation error is given as

$$r(k) = e_n(k) + \lambda_1 e_{n-1} + \dots + \lambda_{n-1} e_1(k) \quad (3.4.17)$$

where e_{n-1}, \dots, e_1 are the delayed tracking errors and λ are constant selected matrices.

With use of the output from the RBF network, $\hat{f}(x)$, and the estimation error $r(k)$ the control signal is found as

$$u(k) = x_{nd}(k+1) - \hat{f}(x(k)) + k_v r(k) - \lambda_1 e_n(k) - \dots - \lambda_{n-1} e_2(k) \quad (3.4.18)$$

where $k_v \in R^{m \times m}$ is a diagonal gain matrix.

Simulations with a 2 link manipulator were in this paper conducted. The results showed that the pruned network gave better results than the unpruned network when a disturbance was present.

3.5 Regularization

Regularization is originally developed by Tikhonov in 63 for dealing with ill-posed problems. However it is often mentioned in the same setting as pruning since regularization reduces the number of free parameters for the network to choose and

is thus suited for avoiding overfitting. The network size is not directly reduced though, so regularization does not exactly fall under the group pruning.

Regularization happens during the training of a network. The training usually include minimizing a cost function, E , and to include regularization a penalty parameter or regularization term, Ω , is added. A new error function for the network is then found as, $\tilde{E} = E + \lambda\Omega$. Here λ is a small constant that determines the fraction of the penalty term to add. This is often referred to as the *regularization factor*. The regularization term Ω may be defined in different ways depending on the type of regularization used.

One of the most common regularization methods is called *Bayesian Regularization* and can for instance be found in [16].

Here the training phase is an optimization task that minimizes a cost function given as

$$F = \alpha E_W + \beta E_D \quad (3.5.1)$$

where E_D is a sum square error between the network output and the desired output and E_W is a regularization term. Both are given as

$$E_D = \sum_{j=1}^M (d_j - y_j)^2 \quad (3.5.2)$$

$$E_W = \sum_{i=1}^W w_i^2 \quad (3.5.3)$$

where d_j is the desired output, y_j is the actual output, M is the dimension of the vector d , w_i are the weights and W is the total number of weights and biases in the network.

The parameter α determines the importance of the exactness in estimating the training data while β is to ensure that the approximation is smooth. A classical training version only has the first term and setting a large α compared to β would result in a smaller training error and a network closer to a network obtained without regularization. Choosing a large β instead would result in smaller weights and smoother network output.

Finding the best values for the α and β are thus the significant problem for regularization and with Bayesian Regularization these two factors are found with Bayesian probability. The interpretation of neural networks here is that optimizing the weights corresponds to increasing the probability

$$P(w|D, \alpha, \beta, A) = \frac{P(D|w, \beta, A)P(w|\alpha, A)}{P(D|\alpha, \beta, A)} \quad (3.5.4)$$

where w is the weight coefficient vector, D is the training data, A is the structure of the neural network, $P(D|\alpha, \beta, A)$ is a normalization element, $P(w|\alpha, A)$ describes the information on the values of the weight prior to introducing the training data and $P(D|w, \beta, A)$ is the probability of obtaining the established response of the network for suitable inputs depending on parameters of the network.

Next the noise in the input data and the probability of weight distribution are assumed to be gaussian and the distribution of α and β to be uniform. Then the equations describing the parameters which minimizes the cost function are found as

$$\alpha = \frac{\gamma}{2E_W(w_{MP})} \quad (3.5.5)$$

$$\beta = \frac{M - \gamma}{2E_D(w_{MP})} \quad (3.5.6)$$

where w_{MP} is the minimum point of the objective function and

$$\gamma = W - 2\alpha \text{trace}(H)^{-1} \quad (3.5.7)$$

where H is the Hessian matrix of the cost function.

The parameter γ means an effective number of parameters of the neural network while W is the total parameters in the network. Thus this regularization will give a reduction of $2\alpha \text{trace}(H)^{-1}$ in effective network parameters.

Chapter 4

Weight Magnitude and Neuron Output Pruning

4.1 Introduction

In this chapter two simple pruning schemes with low computational cost will be proposed. First a method called *Weight Magnitude Pruning* which prunes based on the magnitude of the weights in a network is described. The second method has been called *Neuron Output Pruning* and is based on the output from the activation functions in the hidden neurons.

As mentioned the focused has been on finding a small network with good enough approximation and generalization ability. A small network is a faster network and would be preferred over the very accurate network which perhaps not is able to give an output fast enough to be used in a regulator.

The output of a RBFNN with only one output node is given as in equation (2.2.2) and repeated here

$$F(x) = \sum_{i=1}^N w_i a_i(x) \quad (4.1.1)$$

where there are N units in the hidden layer, w_i is the weight between the output neuron and hidden unit i and $a_i(x)$ is the output of the activation function in unit i to input x .

From (4.1.1) it is possible to see that the output of the neural network is a sum of the contribution from each hidden neuron. This contribution depends in general on two factors, 1) the magnitude of the weights, and 2) the output of the activation functions. If one or both of them are zero for a hidden unit i the

total contribution of that neuron is zero and thus unit i can be removed from the network without affecting the output of the network.

The problem is to find those neurons with a small contribution that not are necessary for the performance and generalization ability of the network. One way of finding these may be to look at the same two factors, the magnitude of the weights and the activation function output. For a unit with small value for one or both of these factors the contribution will be small and hence that neuron can be removed.

The networks taken into consideration here are radial basis function (RBF) neural networks with unity in the weights connecting input and hidden layer and changeable weights between hidden layer and output layer. As mentioned here is only considered networks with a single output node. It should however not be a large problem to extend the methods to having several output nodes.

4.2 Weight Magnitude Pruning

This pruning method is based on the Weights Power Method proposed by Hagiwara in [8].

The idea here is to remove the weights with a small magnitude which would make a neuron give little contribution to the final network output. During training the size of the weight connecting a hidden unit to the output neuron is found and this magnitude can be viewed as a measurement of the importance of that unit. If the training algorithm finds that a certain neuron not is important this neuron will be given a small weight. If the neuron on the other hand is found to be very important the weights will be given a large value to increase the contribution of that exact neuron. By removing some weights the remaining ones will be strengthened when the network undergoes more training and increase the contribution of the remaining neurons.

It should be noticed that since the networks taken into consideration in this report only have one output neuron it will be equivalent to remove a single weight and a neuron.

4.2.1 When to start pruning

When the network starts to learn the mapping between inputs and outputs the weights change less and less until they converge into a fixed value.

Pruning is done based on these weights so it is absolutely crucial that there have been enough training first. If the pruning should start while the weights are

far from set it is very likely that some incorrect ones are removed. The weights experience greatest change in the beginning of the training and only smaller changes at the end. Hence it is reasonable to believe that pruning can be started before the weights have converged into their final value as long as they only are doing smaller changes. This may reduce the total time of training and pruning.

4.2.2 Pruning

Now the problem is to decide what value is small enough for the magnitude of the weights to prune them. This value will vary from task to task and thus it can not be found an optimal threshold valid for every problem.

Being able to remove several weights in the same pruning instead of the slower version of deleting them one by one is desirable. This can be done by using a certain percent of the L^2 norm of the weight vector for finding the threshold of what weights that can be pruned. Since the pruning begins when the weights only are experiencing minor changes the L^2 norm of the weights will almost have converged. The L^2 norm of the weights is a measurement for the total strength of the weights in the network.

Simulations supported that the norm actually converged quite fast and stayed at approximately constant for the rest of the training when the weights did the final small changes. From tests with removing neurons from a network it was also found that this norm stayed constant after pruning and retraining at the same value. When removing some weights the remaining weights were increased as to match the removed ones. From now on the L^2 norm of the weight vector will be referred to as the norm.

Now it is proposed to use a certain percent of this norm as a threshold for the weights that are small enough to be removed. This threshold is given as

$$pW_{th} = \frac{\| W \| pW_{\%}}{100} \quad (4.2.1)$$

where $\| W \|$ is the L^2 norm of the weight vector and $pW_{\%}$ is the percent that the pruning threshold is of the weight norm. The only value to be chosen now is the percent, $pW_{\%}$. Finding this should be less problem dependent than setting the threshold, pW_{th} , directly.

4.2.3 After Pruning

When some weights have been removed the network has to experience more training in order to strengthen the remaining weights. Exactly how the network will

behave after some of the weights have been removed is not possible to predict, and the pruning should be done over several periods. Due to the surrounding weights all being removed a neuron may become much more important than before the pruning. So after pruning and retraining the network should be pruned again. This time with a higher threshold due to the fact that the remaining weights have a higher magnitude than before pruning.

Pruning and training is then repeated for some periods, one period is pruning and training once each, with the percent increasing a bit each time.

4.2.4 When to End Pruning

With a growing threshold some kind of stopping criteria must be given in order to avoid pruning the networks too much. A simple way is to specify a number of times the threshold in percent can be made larger. When this threshold becomes a constant there will not be pruned any more weights as the remaining ones have their magnitude increased.

4.3 Neuron Output Pruning

This pruning method is based a definition from [2] that was used together with weight variance to prune a self-growing radial basis function network.

The importance of a neuron is also found by looking at where that neuron is placed compared to where the inputs occur. Due to the localization principle units in remote areas from the inputs will have a small activation function output.

Thus here a sum of an activation function output ratio for the previous outputs of a neuron to different inputs is proposed to use. If this ratio sum is small for a neuron compared to other neurons after training this node is not close enough to the inputs and hence may it be pruned. The activation function output ratio sum is defined as

$$os_i = \sum_{j=1}^T \sigma_i^j \quad (4.3.1)$$

where T is the number of inputs and σ_i^j is the activation function output ratio for neuron i to input j defined as in [2]

$$\sigma_i^j = \left| \frac{a_i(x)}{a_{max}} \right| \quad (4.3.2)$$

where x is the input, $a_i(x)$ is the activation function for neuron i and a_{max} is the maximum activation function output for that given input.

4.3.1 When to start pruning

It is very important not to start pruning before enough different inputs have been fed to the network. This is because neurons are removed if they compared to other neurons are far away from the previous inputs. So in order to be sure that the correct units are removed the whole input space has to be visited. It is however not necessary that the training is complete before pruning as long there have been enough different training inputs to explore the input space first.

4.3.2 Pruning

When the training has done enough iterations the network can be pruned based on the hidden node output ratio sum. Again to remove many units at the same time is desirable. This is done by specifying a percent of the maximum output ratio sum as a threshold. All the nodes with an output ratio sum smaller than this threshold will be pruned.

The actual threshold, pO_{th} , is then given as

$$pO_{th} = \frac{os_{max} * pO_{\%}}{100} \quad (4.3.3)$$

where os_{max} is the highest output ratio sum and $pO_{\%}$ is the pruning threshold percent.

Since it not is likely to be any changes in the activation function output when removing some neurons the percent threshold is not growing as with the weight pruning. This will probably result in all the removable neurons will be deleted in an early stage of the pruning. Thus the whole pruning process will quickly give a smaller network and with fewer periods than the weight based pruning needed.

4.3.3 After Pruning

After pruning more training is very important as for weight magnitude based pruning. This is because pruning is started before the first training is complete and the weights have not converged to their optimal values. Also more training is necessary due to removing some neurons will probably cause the remaining weights to have another optimal value than for the initial network.

4.3.4 When to End Pruning

Any specific stopping criteria is not necessary here since the threshold is close to constant all the time and the output ratio will not experience any large changes when other neurons are pruned.

4.4 Combined Weight Magnitude and Neuron Output Pruning

The two proposed pruning methods, weight magnitude pruning and node output pruning, removes units based on different criteria and it can thus be assumed that they not always choose the same neurons to prune. In this section two different pruning methods that combines the weight based and the node output based pruning criteria are described.

4.4.1 Both Pruning Criteria Met

Choosing the units with both small weight magnitude and a low output ratio sum should be the safest way to find which neurons to removed. If both criteria are satisfied for a unit then this neuron is completely unnecessary for the network and can be removed.

Pruning is done as for the two methods described earlier. Before the first pruning the network experience enough training and is trained again after being pruned. The weight percent threshold grows a bit for each pruning so there would be the same numbers of periods here as for only weight magnitude pruning. After the pruning there should be a last training for the weights to converge before the final network is obtained.

4.4.2 Only One Pruning Criteria Met

A little less strict pruning method is to prune those units that satisfy either the small weight magnitude criterion or the low output ratio sum criterion.

Using both the pruning methods separately at the same time will with the right thresholds probably give the smallest network still able to give good performance. This is because the output ratio sum pruning removes the units far away from the inputs while the weight pruning remove some of the neurons close to the inputs but unnecessary due to the area being to densely populated with units.

Pruning with both methods separately in each period would probably be the slowest pruning scheme. Both pruning methods however are not computationally

complex. If the final network can be created smaller and still maintain accurate approximations then it could be an advantage to do this.

4.4.3 Mixed Method

The learning controller implemented later in this paper has one RBF network to learn each of the elements in the dynamic matrices as done in [7]. This gives many smaller networks with different properties and thus would pruning them with different threshold or method perhaps give better result than using the same for all of them.

4.5 General Algorithm

Here is a very simple general pruning algorithm that goes for all the proposed pruning methods in this chapter.

1. Initialize a bigger RBF network than assumed necessary
 2. Train the network until the approximation error is satisfyingly small
 3. for $i = 1 : n\text{Periods}$
 - Calculate threshold
 - Prune with the calculated threshold
 - Retrain the network
- end

4. Use pruned network

Chapter 5

Simulations with Pruning for Cross Function Learning

5.1 Introduction - Cross Function

In this chapter all the described pruning schemes from the previous chapter are tested on approximating a cross function. These simulations are the first tests done with the proposed Weight Magnitude and Node Output Based Pruning.

The cross function learned here is a complex function and the network needed for this task is very huge. The network sizes here are not possible in online robot manipulator control and these simulations are done in order to show the concepts of the different pruning methods and to test how they work. By including figures of the estimated cross function given from a pruned network it will be more visible where the different units are removed from.

The cross function is given as

$$z = \max \left[e^{-10x^2}, \max \left[e^{-50y^2}, 1.25e^{-5(x^2+y^2)} \right] \right] \quad (5.1.1)$$

and in figure 5.1 below is a figure of the function.

In this chapter the approximation error for the network is found by using the *Root Mean Square (RMS)* function to calculate the error between the actual and desired output for the network.

The simulations are in done Matlab. The RBF networks used in these simulations and the initial code for the cross function are made by Sigurd Fjerdings.

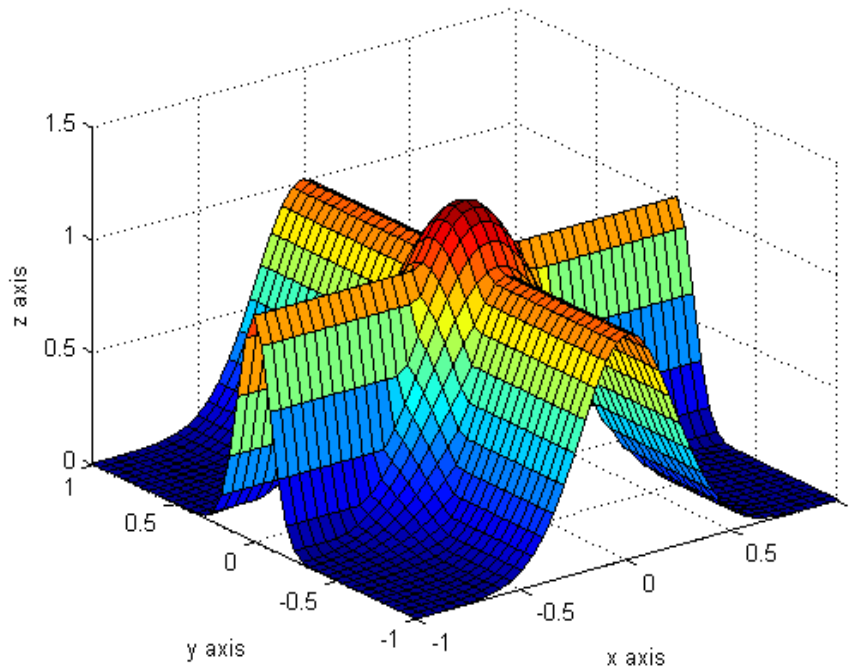


Figure 5.1: Cross Function

5.2 RBF Network

For this task a radial basis function neural network with two inputs and one output is used. The inputs are values for a given point in x-y space and the belonging z value for this point on the cross function is the output of the network.

In chapter 2 it was described the RBF networks used here and their training method.

Choosing the number of neurons in the hidden layer and the size of the training set needed are two of the major problems with neural networks. When using pruning the network has to be large enough and probably also a bit larger than necessary. In this section some results for the cross function when using different sizes for the hidden layer and training set are shown.

5.2.1 Size

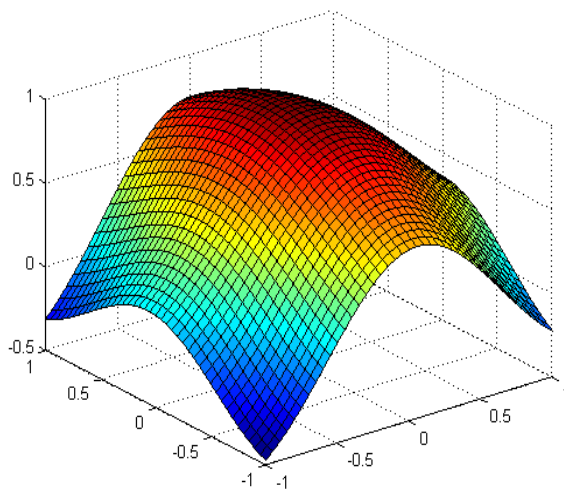
The hidden neurons for all the networks mentioned below are placed in the interval $[-1, 1]$ in both x and y direction and the networks are trained with 40 000 iterations. This is the same as saying the training data set consists of 40 000 samples.

Too Small Network

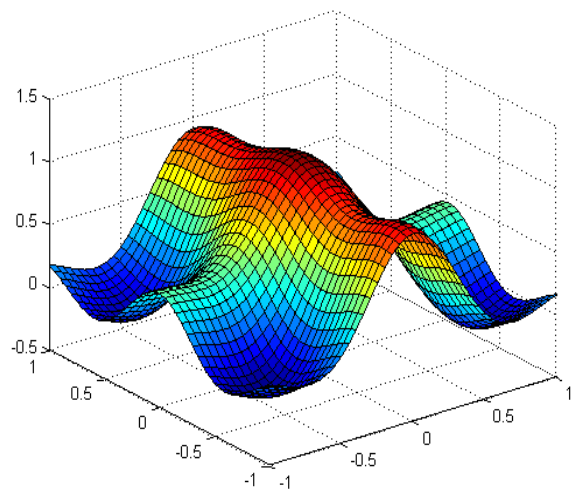
In figure 5.2a there was a distance of 1 between the centres which gave only nine hidden neurons in the network. As the figure shows this network is not even close to approximate the cross function correct and the RMS approximation error was 0.26936.

With a distance of 0.5 between the units in the hidden layer the resulting network has 25 hidden nodes and the resulting approximation of the cross function can be seen in figure 5.2b below. Now it is possible too see some contours of the cross function. Still it is a very poor estimation and the approximation error here was 0.14353.

Placing the hidden nodes with only 0.1 between them gave 441 hidden neurons and this network gave an approximation error of 0.010417. From figure 5.3 it can be seen that the approximated function now looks like the original cross function. However it is far from perfect.



(a) 9 Hidden Units



(b) 25 Hidden Units

Figure 5.2: Too Small Networks

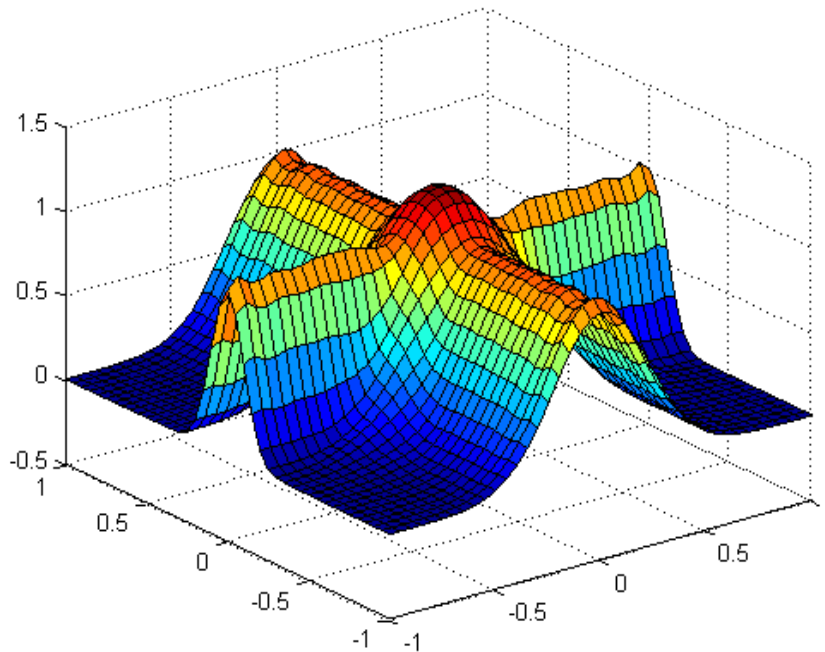


Figure 5.3: Too Small Network 2

In general a too sparsely populated hidden layer for a RBF neural network will give a flat graph and the network will not be able to approximate anything accurately. For the radial basis function network used here the spread of the gaussian function is the same as the distance to the next neuron which with too few neurons will give a very flat activation function. This also contributes to the estimation being even flatter.

Appropriate Size

When using 0.01 as a distance between the hidden nodes the network had a total of 40 401 neurons. This gave an approximation error of 7.0252×10^{-6} and the cross function is estimated perfectly as shown in figure 5.4.

Too Large Network

As mentioned in the introduction one of the problem with choosing the correct size is overfitting. This happens when the network is created too big and the

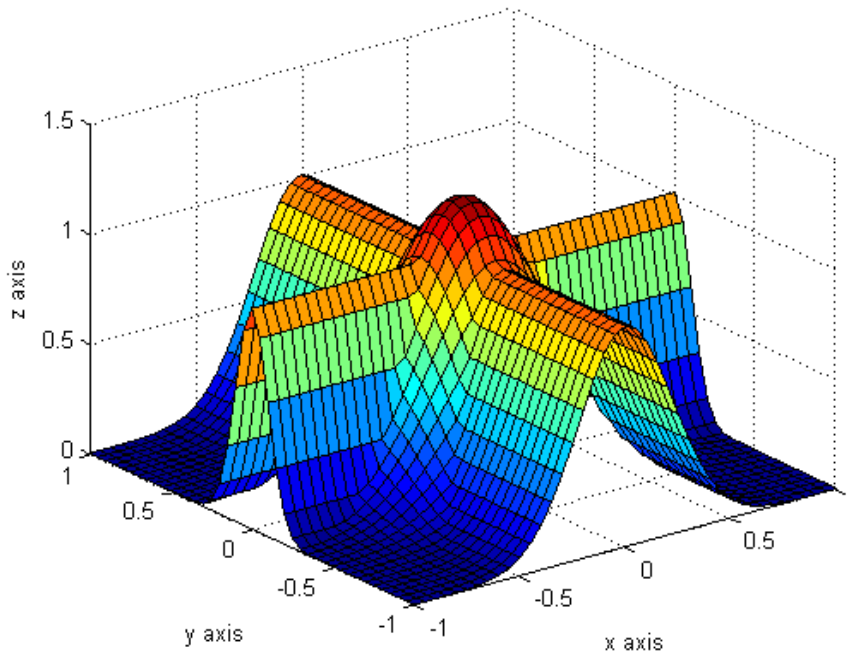


Figure 5.4: Appropriate Size

generalization ability for the network is reduced leading to poorer approximation result. This was however not easy to achieve for the cross function. Using 0.005 as distance between units gave a perfect figure with an error of 1.2983×10^{-5} and the number of hidden neurons was then 160 801. This error is a bit larger than the one for 40 401 hidden units that had 7.0252×10^{-6} as an approximation error. However not a large different and there are no visible errors on the resulting figure.

A network with 4 004 001 neurons in the hidden layer and a distance of 0.001 between the units gave an approximation error of 1.5122×10^{-5} . This is a bit higher than for the last network. Not much and the error is still very good. This network is extremely large and these calculations took many hours. Thus it is not tried with an even bigger network and it can only be assumed that overfitting will happen as more neurons are added to the hidden layer. This is supported by the fact that the approximation error got slightly worse.

An example of a network that became overfitted can for instance be found in [13].

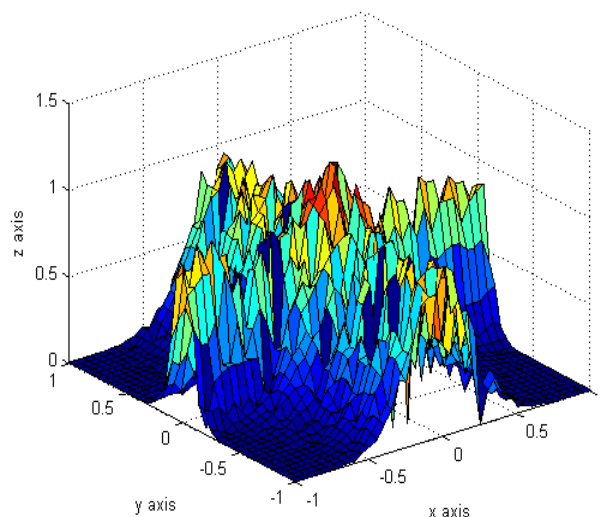
5.2.2 Training

It is crucial that the network experience a training set with enough varied training samples. If there are no training samples from one area of the cross function this area will not be learned due to the localization principle and the network will not be able to give any good approximation of this area.

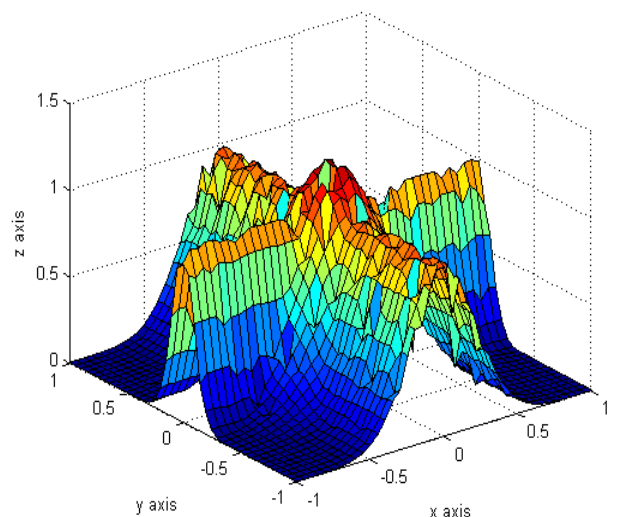
Here coordinates in x,y space is randomly chosen and used to train the network. One training sample is a coordinate, (x,y), and it takes one training iteration to train with one training sample.

All the networks in this section are in the interval $[-1, 1]$ and there are 40 401 hidden neurons.

Too Little Training



(a) 5000 training iterations



(b) 10 000 training iterations

Figure 5.5: Networks Trained too Little

In figure 5.5a there has only been 5000 training iterations. This obviously not is enough training for this network to learn the cross function. It can be seen that the weights not have been sufficient incremented to give high enough z-values as correct outputs. The approximation error is 0.16574.

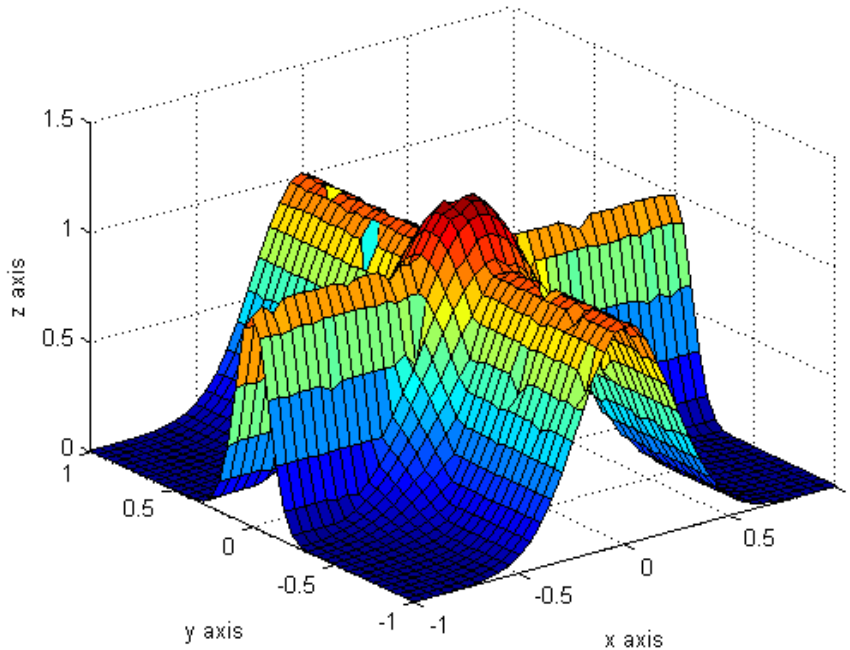


Figure 5.6: Networks Trained too Little 2

When the training set consists of 10 000 training samples the resulting network will approximate the cross function as shown in figure 5.5b. Here it can be seen that the function is almost correct in some areas while other areas have "holes". The approximation error was 0.052378.

The approximated cross function in figure 5.6 is from a RBF network trained with 15 000 iterations and the approximation error here is 0.01758. The function now looks almost correct but still some areas are not enough trained and have too low output values.

With 30 000 samples in the training set the function is correct when looking at the figure and the error for the approximation is 0.00023843.

In the previous section regarding the size of the network it was used 40 000 training iterations since this gives a smaller approximation error, 1.5121×10^{-5} .

This section found that the differences between the approximated cross function when the network has too little training or too few hidden units are huge. When there are not enough training samples the approximation is not smooth and there are "holes" in some places where there have been little training samples from.

Other areas are close to perfect since these areas have experienced enough training.

Too much training

Too much training for a network can result in the network overfitting to the training data set almost in the same way as a too large network would do. This seems however not to be possible with this network as the error only decreases.

After 80 000 training iterations is the approximation error 7.9875×10^{-12} , while the network trained with 150 000 training samples had an error of 4.2675×10^{-17} . Even after training with a training set of 300 000 samples the error was still decreasing. It was then 4.003×10^{-17} .

Since using this large training data set for a network with as many as 40 401 hidden neurons take a very large amount of time to finish it is here not tried to with an even larger training set. For more on overfitting can [13] be seen.

5.2.3 Network Used Further

The network used further in the simulations with the cross function is the network with 40 401 hidden nodes placed in the interval $[-1, 1]$ with a distance of 0.01 between them. Estimations from this network are perfect but the number of hidden units is very large. Thus it is desirable to see if this network that can give a very good result can be made smaller and still give an accurate approximation.

Since it not is necessary for the weights to have converged before pruning the number of training iterations is smaller than 30 or 40 000 as the good networks in this section were trained with. The number of training samples used will be mentioned in the next section.

5.3 Results

It is absolutely necessary that the network has experienced enough training before being pruned and here it has been used a first training phase for the network which ends when the approximation error is smaller than 0.01. This happens between iteration 16 000 - 19 000 usually and is only checked every 1000 iteration in order to avoid too much computation.

A period for the network is a pruning phase followed by a retraining phase. So first the network is trained to a RMS error smaller than 0.01 and then starts the

first period with pruning and retraining.

5.3.1 Only Weight Magnitude Pruning

Here done the first simulations with the proposed weight magnitude pruning from the previous chapter is conducted. The threshold for weight pruning is given as a percent of the L^2 norm of the weight vector as $pW_{\%}$ in equation 4.2.1. So when it here is specified a chosen threshold it is always given as this percent. The actual value for the pruning threshold is then found from this threshold as in equation 4.2.1.

When describing the weight magnitude pruning it was mentioned that the pruning should be done over several periods in order to avoid removing some necessary weights. For this it was needed a growing threshold.

First time the threshold in percent, $pW_{\%}$, is the same as the starting threshold. Second time the threshold is $pW_{\%} * 2$, third time the threshold becomes $pW_{\%} * 3$, and so on until the threshold for pruning time N is found as $pW_{\%} * N$.

Pruning Results for Weight Based Pruning					
	<i>Final Size</i>	<i>Approximation Error [RMSE]</i>	<i>Threshold [%]</i>	<i>Periods</i>	<i>Retraining</i>
1	10 880	0.0018618	0.015	5	15 000
2	12 392	0.0098295	0.02	3	15 000
3	17 225	0.00048589	0.005	5	15 000
4	24 600	6.66692×10^{-5}	0.001	5	15 000
5	34 901	3.1553×10^{-7}	0.00005	5	15 000

Table 5.1: Pruning Results for Weight Based Pruning

In table 5.1 some of the different pruning results obtained by using different thresholds and number of periods can be seen. The threshold specified in this table is the starting threshold. Final size is given as the number of hidden units in the final network and the approximation error is found as the root mean squared error. It is also specified the number of training samples used in the retraining phases.

Obtained with weight magnitude based pruning the smallest possible network still able to approximate the cross function such that the figure looks perfect is a network with 10 880 hidden nodes. This is test 1 in table 5.1 and the approximation error for this network was 0.0018618. The starting threshold was 0.015% of the

norm of the weight vector and was increased for each of the 5 pruning times as described above.

The first training ended after 18 000 iterations with an approximation error of 0.0099721. How many weights and neurons that are removed (it is equivalent to remove a weight and a neuron) varies a lot from pruning to pruning. Most of the nodes are removed during the first period. For this network were 21 532 nodes removed in the first pruning, 12 in the second, 5508 in the third, 1388 in the fourth and 1072 in the fifth.

It has been tried to increase to 20 000 training samples for the retraining phases, this gave however no large improvement for the approximation error compared to only using 15 000 samples. Thus it is concluded that 15 000 training iterations is enough for retraining the network when using weight magnitude pruning. Anything less however is found to be insufficient.

From the simulations done with the weight magnitude pruning it was found that using a lower starting threshold and prune over several periods gave better approximations than using a higher starting threshold and prune over fewer periods. This can be seen when looking at test 1 and test 2 in table 5.1. Here the network in test 1 has both fewer hidden neurons and smaller approximation error than the network in test 2. The main difference between them is that the network in test 2 is pruned with a higher threshold over 3 periods while network 1 is pruned with a lower threshold over 5 periods.

Another network was created by using 0.001% as a pruning starting threshold instead of the above mentioned 0.015%. The obtained network gave 24 600 hidden units and an approximation error of 6.66692×10^{-5} as can be seen in test 4 in table 5.1. This approximation error is a lot smaller than the error for the smallest network. However there are also a lot more hidden nodes. Perhaps the most suited network would be somewhere between these two sizes like the network in test 3 in the table. That network has 17 225 hidden neurons and an approximation error of 0.00048589. The smallest possible network from test 1 was very close to the networks not able to give an accurate approximation and hence it could be preferred to use a network with better approximation ability. However, 24 600 hidden nodes as in test 4 or even 34 901 hidden units as in test 5 would result in the computational cost being much larger and therefore not be the best suited network here.

Too Much Weight Pruning

When the network is pruned too much with weight based pruning some of the units that are necessary for the network to give an accurate approximation will be removed. One such an example can be seen in figure 5.7 below. Here the pruning had a starting threshold set to 1% and lasted for 3 periods. The final network had 705 hidden units and an approximation error of 0.10642. From this figure it is possible to see that there have been removed some necessary neurons from the bottom part of the cross function. The function here is completely flat until it suddenly goes very steep up. The upper part of the function is approximated fine.

This shows that having a too high threshold with weight pruning will remove too many units from the parts with a lower estimation value. The center part of the cross functions still approximated correct is the part of the cross function with the highest z-value. All the sides have lower z-values and thus also lower weight magnitudes in order to give a smaller output of the network for those points. Since the function here is estimated to be zero it shows that all units in these areas have been removed.

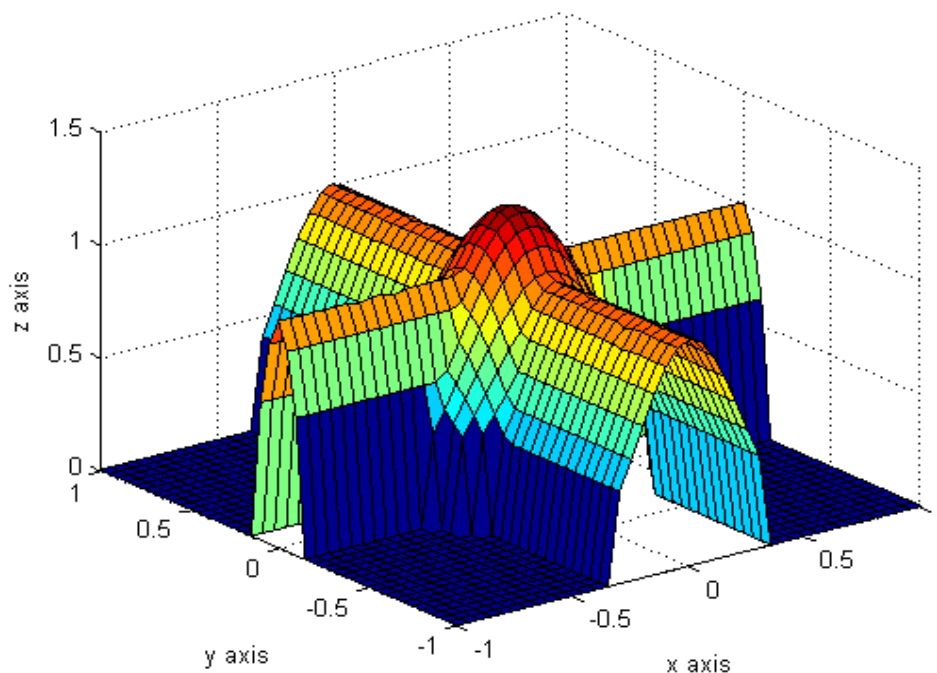


Figure 5.7: Over Pruned by Weight Magnitude Pruning

5.3.2 Only Neuron Output Pruning

In this section the first simulations with the proposed neuron output pruning are done. It is pruned based on the output ratio sum of the activation functions as described in an earlier chapter. Since it is not necessary with increasing threshold for this pruning method there are no need for the same number of periods as with weight magnitude based pruning. Hence this will be a faster pruning method.

It was however found to be necessary with some more training samples in the retraining phases and here training set with 20 000 samples for the retraining is used. There were only removed neurons in the first two periods and thus only 2 periods are needed.

In table 5.2 some results for different pruned networks with neuron output pruning are shown. It can be seen the final number of hidden neurons in the network and the root mean squared approximation error. The threshold is specified in percent of the node output ratio sum. For retraining it is specified the number of training iterations in the retraining after the first pruning and also the number of samples used in the last training of the network.

As can be seen from this table there is a huge difference in the approximation error between test 6 and test 7 where both networks are pruned the same way and have almost the same size. The difference between them is that the network in test 6 has been trained with 30 000 iterations in the final retraining while the network in test 7 was trained with 20 000 samples in the last retraining. Thus there are used different numbers of training samples when pruning with node output based pruning. The initial training still lasts until the approximation error found by RMSE is smaller than 0.01 and not changed anything from weight magnitude training.

Pruning Results for Neuron Output Pruning					
	<i>Final Size</i>	<i>Approximation Error [RMSE]</i>	<i>Threshold [%]</i>	<i>Periods</i>	<i>Retr.(Last Retr.)</i>
1	8 221	0.0001182	7.5	2	20 000(30 000)
2	8 613	2.9779×10^{-5}	6.5	2	20 000(30 000)
3	11 194	1.1263×10^{-5}	5	2	20 000(30 000)
4	14 357	4.6827×10^{-7}	3	2	20 000(30 000)
5	14 581	1.5198×10^{-5}	2	2	20 000(30 000)
6	14 607	5.9554×10^{-9}	1.5	2	20 000(30 000)
7	14 627	0.00023403	1.5	2	20 000(-)

Table 5.2: Pruning Results for Neuron Output Based Pruning

As can be seen in the table 5.2 there are two results with the threshold 1.5%, test 6 and 7, and the final network sizes are a bit different. This is probably due to the fact that the training set consists of randomly chosen coordinates and thus the different areas are not visited the same amount of times for each training. This gives that the output ratio sum is a bit different from test to test and some of the units removed are not the same each time. However most of the pruned neurons are the same each simulation and the final networks are almost equal.

The smallest network obtained with node output based pruning still able to approximate the cross function with no visible errors had 8 221 hidden units and a final approximation error of 0.0001182. This is test 1 in table 5.2 and the pruning threshold was then 7.5%. The initial training ended after 17 000 iterations with an error of 0.0094069. There were pruned 30 943 nodes in the first period and 1237 nodes in the second.

From the same table it is visible that using 6.5% as a threshold gives a bit bigger network and a much smaller error. It can also be seen that the approximation errors when the networks had close to 14 000 hidden units are almost the same as the networks with 20 000 - 30 000 hidden units after weight pruning.

When using node output pruning the preferred network is probably the network in test 2 with 8613 hidden units and an approximation error of 2.9779×10^{-5} . This network was obtained with a pruning threshold of 6.5% and the pruning was done over two periods.

Too Much Node Output Pruning

When the network is pruned with a too high node output threshold the resulting networks do not give the same result as when they are pruned with a too high weight magnitude threshold. This is visible when looking at figure 5.8. The final network here has 1415 hidden neurons and the approximation ends up having some "holes" in the function where the given approximation is zero. For this approximation the RMS approximation error was for 0.23241 and the pruning threshold was 20%.

As can be seen from the figure 5.8 it has here been pruned too many neurons from the whole network. No larger area gives a correct estimation and there are no large areas where the estimations given are zero. Even in the areas where the z-value is small there are still some output from the network. This is because there have been inputs in these areas and hence the activation function output is high for the nodes in this area even if the z-value is small. A smaller output is obtained with having a smaller weight magnitude as was visible when the networks were

pruned too much with weight based pruning.

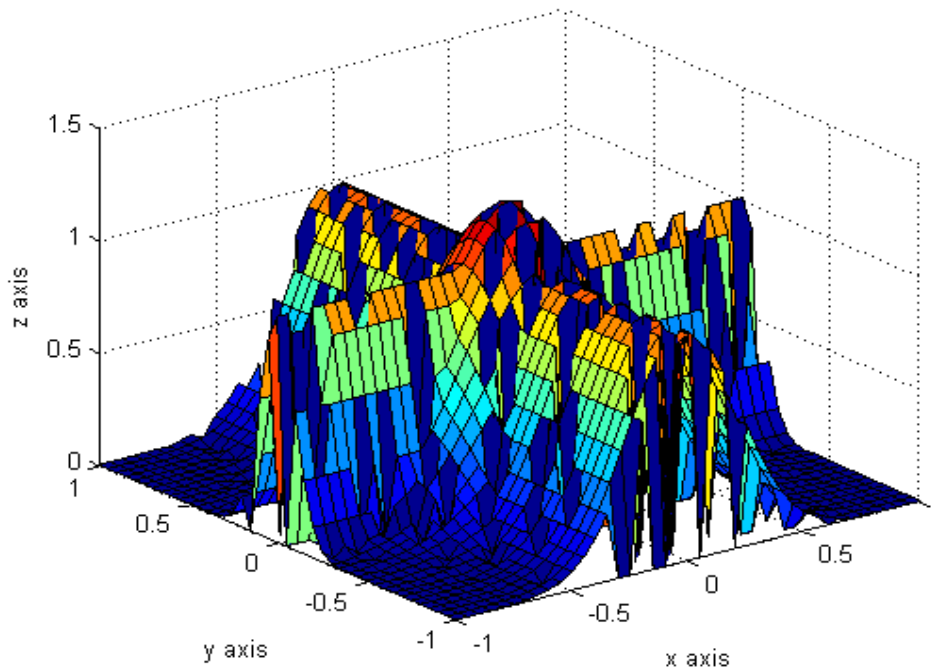


Figure 5.8: Over Pruned by Node Output Pruning

5.3.3 Both Pruning Methods - Separately

In this section pruning with both weight magnitude and node output pruning at the same time is tested. First the network is pruned by node output ratio pruning and then by weight magnitude pruning in the same pruning phase.

Node output pruning quickly removes many neurons in the areas with little or no inputs. However it is not so good at the areas where more inputs appear. If these areas have more nodes than required the pruning these of unnecessary units with only node output based pruning will be poor. Weight magnitude pruning is not only based on the distance to the inputs so here it is tried to prune these areas with weight pruning when output pruning is done.

Some results obtained when using both node output and weight magnitude pruning are shown in table 5.3. Again the final number of hidden units and the approximation error found by the root mean square function are specified. The

Pruning Results for Both Methods in Same Pruning						
	<i>Final Size</i>	<i>Approx. Error [RMSE]</i>	<i>Thresh. [%] Weight(Output)</i>	<i>Periods</i>	<i>Retraining (Last Retr.)</i>	<i>Units Removed Weight(Output)</i>
1	7269	0.00088648	0.015 (7.5)	3	20 000 (30 000)	2105 (31 027)
2	8622	0.0011358	0.02 (6)	3	15 000 (-)	3246 (28 533)
3	10 884	0.0019155	0.015 (0.01)	5	15 000 (-)	23 037 (6480)
4	10 889	0.0018595	0.015 (0.01)	5	20 000 (30 000)	23 056 (6456)
5	14 104	0.00048173	0.01 (0.1)	3	20 000 (30 000)	14 257 (12 040)

Table 5.3: Pruning Results for Both Methods in Same Pruning

threshold shown as the starting threshold for the weight based pruning and the constant threshold for node output pruning are in percent. It can be seen if it has been used different numbers of training iterations in retraining and for the last retraining. Also in this table how many of the neurons that are pruned by weight magnitude pruning and how many removed by node output pruning are shown.

From table 5.3 it can be seen that the smallest network has 7269 hidden units and is pruned for 3 periods with 0.015% threshold for weights and 7.5% threshold for neuron output ratio. The approximation error for this network was 0.00088648. Most of the removed units were pruned by node output pruning.

It can from this table also be seen that deleting most of the units with neuron output pruning gives a better approximation error than removing most of the units with weight magnitude based pruning. This is visible when comparing for instance test 1 and test 3. For test 3 most of the units are removed by weight pruning while for test 1 most of the nodes are removed by output threshold. The result is that the network in test 1 is smaller in size and also has a smaller error.

The network in test 3 in table 5.3 with 10 889 hidden units obtained with 0.015% weight threshold and 0.01% output threshold is approximately the same network that was obtained with only 0.015% weight pruning over 5 periods. That network had 10 880 hidden units and an approximation error of 0.0018618. Thus the networks were almost equal in both size and final approximation error. This is because most of the weights are removed by the weight pruning and hence the resulting network is close to the one where all the weights are removed by weight

pruning.

For the situations where many of the units are pruned with neuron output pruning it was found to be better with 20 000 samples in the retraining sets and 30 000 training samples in the final training as with only node output pruning. The training amount was probably what gave the network in test 1 better approximation result than the larger network in test 2. On the other hand it was found to be unnecessary to have more than 15 000 training iterations when most of the neurons were removed by weight magnitude pruning. This can be seen if comparing test 3 and test 4 in table 5.3.

It has been shown that it is in fact possible to create a smaller network when combining the two different pruning methods in this section. Again it is found that weight magnitude and node output not remove the same neurons. Hence pruning the network with both of the methods gave a smaller network that still approximated the cross function with no visible errors.

The best network for this situation was probably the smallest network in test 1 which also had a small approximation error.

Too much pruned

In figure 5.9 it can be seen the resulting approximation of the cross function when the network has been pruned to much with the combination of weight magnitude and node output pruning mentioned in this section. Here the bottom of the function is completely flat as when the network is pruned too much with only weight magnitude based pruning. There are also some large holes in the figure as when the network is over pruned by only node output ratio based pruning.

The network that gave this estimation had 1645 hidden neurons and an approximation error of 0.10251. It was pruned for 3 periods with a starting weight threshold as 0.2% and a node output threshold as 20%. There was used 15 000 iterations in the retraining and 30 000 iterations in the final training.

5.3.4 Both Weight Magnitude Threshold and Neuron Output Pruning Threshold met

In Chapter 4 it was mentioned the safest way should be to remove only those units that had both a small weight connecting to the output layer and little output of the activation function. If an unit met both the criteria for weight pruning and for node output ratio pruning this unit would be completely unnecessary for the network and thus it could be removed.

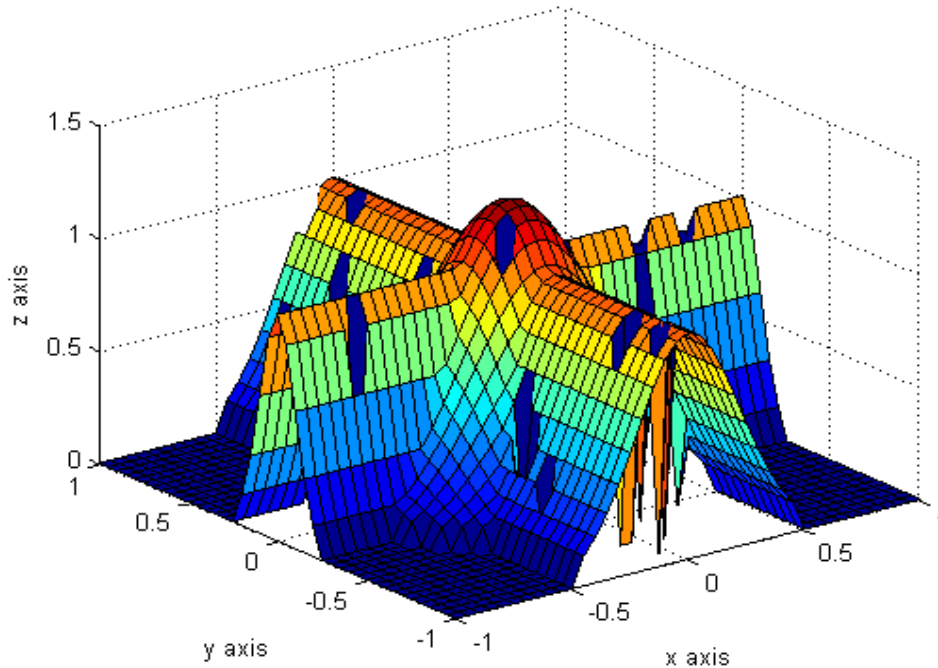


Figure 5.9: Over Pruned by Both Weight and Node Output Pruning

However trying this in simulation gave poor results. The units the two different pruning methods wants to remove are often not the same and often they do not find them at the same time. Thus fewer neurons are pruned when both criteria have to be met and the final network becomes larger. In addition the approximation error was much worse than just using one of the methods.

This was tried in several ways and all the results showed it was much better to use only one of the methods or both but still separately as described in the last section. This version is also slower than just using one of the pruning methods. Hence this pruning scheme is not used any further.

5.4 Discussion

In this section the different results obtained in this chapter are discussed. First the findings regarding the influence of size and training amount for the networks are commented followed by a discussion on the results from the proposed pruning

methods.

5.4.1 Training, Size and Approximation Error

Finding the size of the starting network is an important part of creating a radial basis function neural network. As it was shown in this chapter a too small network will not give any good result. The approximation became flat and had poor accuracy. With pruning a number of hidden units to begin with is easier to find due to it not being necessary also to think about making the network smallest possible. As long as the networks are created with more hidden units than needed, the final networks end up with approximately same size independently of their starting size if they are pruned with the same threshold. Given that the networks also have experienced the same amount of training they will have almost same approximation errors.

The number of training iterations is also an important factor and the network needs a certain amount of training before it can be pruned. Since the focus in this report is to minimize the computational speed for the network while still maintaining a good approximation it was in this chapter tried to find when the network earliest possible could be pruned. The shorter amount of time with a larger network would give less computations and hence be faster in use. It was here found that the approximation error gave a good measurement of when the network had been trained enough and 0.01 was an appropriate number for the root mean squared estimation error to stop the initial training at. Using the approximation error for when to end the first training stage should also be less task dependent than specifying the number of training iterations needed.

Use of one of the three pruning methods tested in this chapter will make it easier to find a starting network. The problem here is to find the correct and best possible thresholds used to prune with. Since the thresholds are specified as percent it should be a little less task dependent and easier to find. However in the end the optimal threshold for a given problem can only be found by testing on the network in simulations.

It can be argued to only do several tests to find the smallest possible network without pruning. However then would also the problem of where to place the centres become important. This probably is an even larger problem than just choosing the number of hidden units. Those two problems are also closely linked. When pruning is implemented the network can be created on a regular lattice and the final network will have the number of nodes required placed in the correct areas.

The estimation errors in this chapter are often not very good and much smaller errors are possible to obtain. However the focus here has been on creating the smallest possible network able to perform good enough. The smaller the network the faster the computations are. Computational speed is very important if radial basis function networks are to be used for online learning in robot manipulator control. Thus it has also been focus on not using more training epochs than required since this also affect the computations in total.

In this chapter the simulations have been for offline training and pruning. Here the estimations given from the network are not used before the network is completely done with pruning and training. When the networks are to be implemented in a robot manipulator controller this however will not be the case as the networks are used online and the estimations will be important from the beginning of.

5.4.2 Pruning Methods

The weight magnitude based pruning method has in this chapter been tested. It was found that implementing this method with an appropriate starting threshold gave a final RBFNN smaller in size and with a satisfyingly good approximation.

Weight magnitude based pruning removes those units that have a connecting weight with a small magnitude. Nodes far away from the inputs will not experience the same amount of weight updating which can be seen from the weight updating law in equation 2.3.3. This contains the activation function output and to an input far away will this be approximately zero due to the localization principle. Also the weights in this report (and very often in other settings) are initialized to zero. Weights starting at zero and experiencing little or no updating will off course stay at approximately zero. Thus weight magnitude pruning will remove those units in remote areas comparing to the where the inputs appear.

However the weights are also an indication of how high the network output to a given input is. From the simulations in this chapter it can be seen that a high z -value gave a very high weight for the neurons that were active in estimation of this z . If the z -value on the other hand was small the active units got weights with a smaller magnitude. This became obvious when looking at the estimated function from networks pruned with a too high threshold. The problem with this method may then be that areas with small desired outputs will be pruned much more than others even if there are just as many inputs in these areas.

Node output ratio pruning removes the neurons unnecessary to the network due to their distance to the inputs. Even if the total output of the network to a given input is small the output ratio can be high for a node placed close to that

input. This is also due to the localization principle. For the cross function in the simulations here the z value in some places was small but the neurons placed in these areas still had a large activation function output. These neurons were thus not removed when the network was pruned by node output pruning. This is desirable since the output in these areas also are important even if they have a smaller value.

From the network over pruned with node output ratio based pruning this could be seen as it was removed too many neurons from the whole network. No areas gave a correct estimation and there were no large areas where the estimations given were zero. Even in the areas where the z -values are small there were still some outputs from the network. Achieving small z -values as outputs are done with having a smaller weight magnitude as could be seen when the networks were pruned too much with weight based pruning.

When comparing the final networks after pruning with either weight magnitude, neuron output or both methods it can be seen that the final networks from weight magnitude alone gave the poorest results. The smallest network here had 10 880 hidden units while neuron output pruning alone could obtain a network with 8221 hidden units. When implementing weight magnitude pruning it was used more periods and it took more time to obtain the final network.

The approximation errors for the weight magnitude pruned networks were also far worse than for the networks obtained with node output pruning. While the network with 8221 hidden nodes from output pruning had an approximation error of 0.0001182 the larger network from weight pruning with 10 880 hidden units had an error of 0.0018618. This is more than ten times the error for a larger network.

The network with 8613 hidden nodes from node output pruning had an error of 2.9779×10^{-5} . If a network from weight pruning was to have an error equally small it was needed more than 24 600 hidden units.

Combining the pruning methods with first pruning the network with node output and then with weight magnitude gave that an even smaller network than the one from only node output could be obtained. The smallest network then had 7269 hidden units with an approximation error of 0.00088648.

When using the combined method it was found that the best results were obtained when most of the units were pruned by node output. This is consistent with the fact that the node output pruning gave better results than the weight magnitude pruning. Only pruning with neuron output pruning gave however in general always smaller approximation error than using the combined version. It could also be pruned one period less when only using node output pruning. Thus the final network would be ready a bit faster than when the combined pruning was used.

Which network that can be concluded on being the best is perhaps dependent on how large network that can be implemented for a given task. If to have the smallest network possible is most important the combined pruning scheme is best suited. If it however can be implemented a bit larger network it would be smarter to use the node output pruning and obtain a network with better approximation ability.

Chapter 6

Online Learning Controller for a 2 DOF Manipulator

6.1 Introduction

After testing the proposed pruning methods, Weight Magnitude Pruning and Node Output Pruning, for offline learning of the cross function the same pruning methods are in this chapter implemented in an online learning controller for manipulator tracking control. Since the networks here are in a different setting with much smaller size from the network used to learn the cross function all the proposed pruning methods are again tested.

The manipulator in these simulations is a 2-dof robot manipulator with revolute joints controlled by a learning inverse kinematic controller to track a desired trajectory. For learning the dynamics of the manipulator several radial basis function networks are used. These dynamics are a part of the inverse kinematic controller and learning for the networks happen online while following the desired trajectory. The online learning consists of training the networks and pruning them. All the networks initially are too big and contain several unnecessary hidden nodes that are found and removed by either Weight Magnitude Pruning, Node Output Pruning or a combination of them.

In this chapter it is calculated the tracking error for the manipulator as the *Sum Squared Error (SSE)* for both joint 1 and joint 2.

For the approximation error of the networks the *Root Mean Square Error (RMSE)* is used as it is with the cross function approximation error. The final estimation error here is only found over the last 300 iterations out of 8795 iterations in total that the simulation takes. This is to check the approximation

error for the networks after they are pruned and retrained. Simulation time is set to be 4π .

Simulations are done by using Matlab and Simulink. The RBF networks used in Matlab are made by Sigurd Fjerdingen and the manipulator model is made by Stefan Pchelkin and Serge Gale. Serge Gale has also made the initial code without pruning for learning the dynamic model.

6.2 2 Dof Robot Model

For a 2-dof manipulator the dynamic equations from equation (2.6.1) can be written as

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} \ddot{q}_1 \\ \ddot{q}_2 \end{bmatrix} + \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} + \begin{bmatrix} g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} \tau_1 \\ \tau_2 \end{bmatrix} \quad (6.2.1)$$

where q is the joint position, \dot{q} is the joint velocity, \ddot{q} is the joint acceleration, $M(q)$ is the inertia matrix, $C(q, \dot{q})$ is the centrifugal matrix, $G(q)$ is the gravitation matrix and τ is the control torque. In figure 6.1 a picture of a manipulator with 2 revolute joints can be seen. This picture is from [18].

6.3 Controller and Desired Trajectory

In these simulations an *Inverse Kinematic Controller* from [18] is used, given as

$$\tau = M(q)(\ddot{q}_d + K_d(\dot{q}_d - \dot{q}) + K_p(q_d - q)) + C(q, \dot{q})\dot{q}_d + G(q) \quad (6.3.1)$$

The gains are as follow

$$K_p = \begin{bmatrix} 250 & 0 \\ 0 & 450 \end{bmatrix} \quad (6.3.2)$$

$$K_d = \begin{bmatrix} 0.5 & 0 \\ 0 & 4 \end{bmatrix} \quad (6.3.3)$$

In figure 6.2 below the trajectory the end effector is to follow is shown. The desired trajectory for each joint separately is shown in figure 6.3.

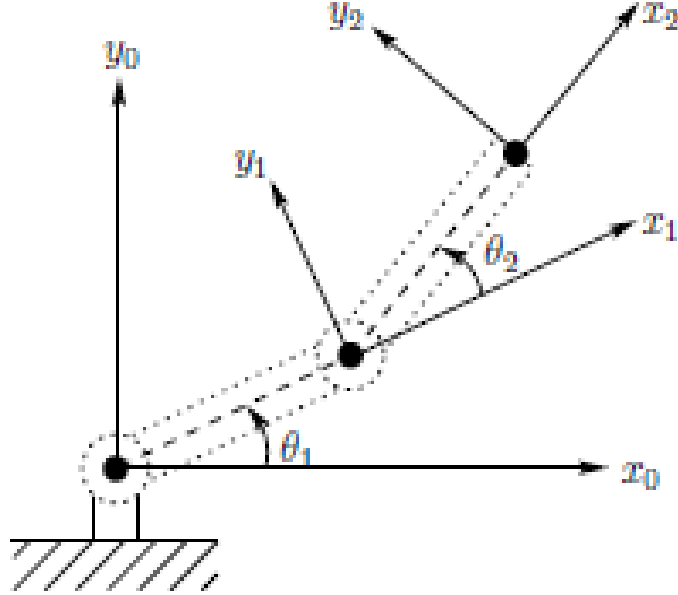


Figure 6.1: 2 DOF Revolute Joints Manipulator

In A.1 by use of Lyapunov theory it is shown that the inverse kinematic controller can be made globally asymptotically stable (GAS) when it is implemented with the model of the manipulator. This assures convergence of the trajectory to the desired one.

For the situation where the dynamics are implemented by RBF networks proving stability and convergence to the desired trajectory is not so easy. While the stability of the standard model-based inverse kinematic controller is well known stability proves for controllers with neural networks are in general lacking in the literature.

In A.2 an attempt to use a Lyapunov Function Candidate to prove stability of the learning inverse kinematic controller is included. It is found that the system with e, \dot{e} where $e = q_d - q$ is state strictly passive when an upper bound on the approximation error from the networks are taken as input. From lemma 6.7 in [17] this system will then be 0-GAS. However 0-GAS is when the input is zero and in this case the input, e.g. the approximation error, is not zero. If the networks are appropriately created and trained it follows from their universal approximation

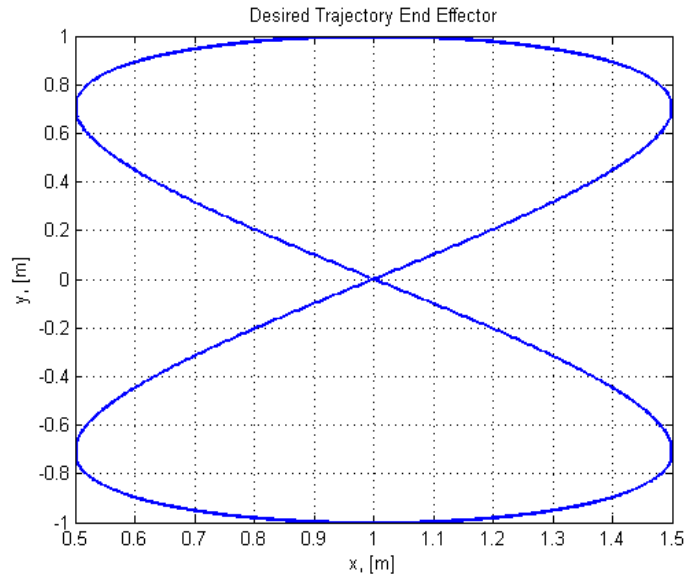


Figure 6.2: Desired Trajectory End Effector

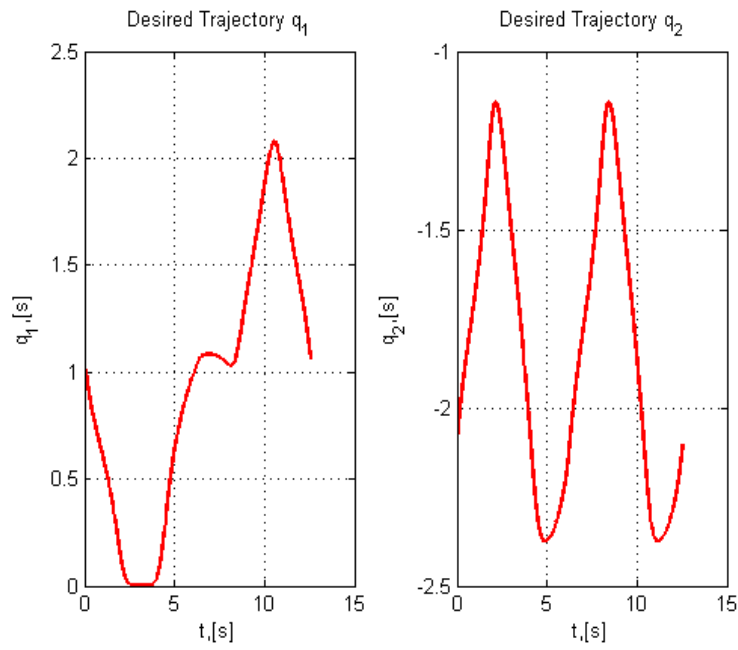


Figure 6.3: Desired Trajectory Joint 1 and Joint 2

ability that this approximation error will be small. Small is still not zero and a proof of stability for this controller should be further investigated.

Here the networks are in addition pruned during real-time control of the manipulator and this can with over pruning cause problems for the controller. Whether these thresholds are suitable or not can only be tested by simulation.

6.4 Radial Basis Function Neural Network

The system matrices $M(q)$, $C(q, \dot{q})$ and $G(q)$ are here learned by the radial basis function neural networks described in chapter 2. Here one network per element in the matrices as described in chapter 2.5 is used.

Comparing to the cross function learning the difference here is as mentioned that the training and pruning are online and thus the outputs from the networks are used in control of the robot from the beginning of the first training.

With notation from chapter 2.5 the estimated inertia matrix can be written as

$$\hat{M} = W_M^T \bullet A_M(x) = \begin{bmatrix} \mathbf{w}_{m11}^T & \mathbf{w}_{m12}^T \\ \mathbf{w}_{m21}^T & \mathbf{w}_{m22}^T \end{bmatrix} \bullet \begin{bmatrix} \mathbf{a}_{m11}(x) & \mathbf{a}_{m12}(x) \\ \mathbf{a}_{m21}(x) & \mathbf{a}_{m22}(x) \end{bmatrix} = \begin{bmatrix} \mathbf{w}_{m11}^T \mathbf{a}_{m11}(x) & \mathbf{w}_{m12}^T \mathbf{a}_{m12}(x) \\ \mathbf{w}_{m21}^T \mathbf{a}_{m21}(x) & \mathbf{w}_{m22}^T \mathbf{a}_{m22}(x) \end{bmatrix} \quad (6.4.1)$$

and the coriolis and centrifugal matrix as

$$\hat{C} = W_C^T \bullet A_C(x) = \begin{bmatrix} \mathbf{w}_{c11}^T & \mathbf{w}_{c12}^T \\ \mathbf{w}_{c21}^T & \mathbf{w}_{c22}^T \end{bmatrix} \bullet \begin{bmatrix} \mathbf{a}_{c11}(x) & \mathbf{a}_{c12}(x) \\ \mathbf{a}_{c21}(x) & \mathbf{a}_{c22}(x) \end{bmatrix} = \begin{bmatrix} \mathbf{w}_{c11}^T \mathbf{a}_{c11}(x) & \mathbf{w}_{c12}^T \mathbf{a}_{c12}(x) \\ \mathbf{w}_{c21}^T \mathbf{a}_{c21}(x) & \mathbf{w}_{c22}^T \mathbf{a}_{c22}(x) \end{bmatrix} \quad (6.4.2)$$

while the gravitation matrix can be written as

$$\hat{G} = W_G^T \bullet A_G(x) = \begin{bmatrix} \mathbf{w}_{g1}^T \\ \mathbf{w}_{g2}^T \end{bmatrix} \bullet \begin{bmatrix} \mathbf{a}_{g1}(x) \\ \mathbf{a}_{g2}(x) \end{bmatrix} = \begin{bmatrix} \mathbf{w}_{g1}^T \mathbf{a}_{g1}(x) \\ \mathbf{w}_{g2}^T \mathbf{a}_{g2}(x) \end{bmatrix} \quad (6.4.3)$$

One network for each element gives a total of ten networks. In the model for the manipulator the element c_{22} is zero and to make things a bit simpler this element has been set to zero and not learned by any network. Thus there are 9 networks.

It is only one output from each network and the number of inputs is varied and based on how many variables the element is dependent on. By variables it is meant q and \dot{q} . For instance m_{11} is only dependent on q_2 and the network for this element has only one input. Element c_{11} is dependent on both q_2 and \dot{q}_2 and hence \hat{c}_{11} has two inputs. Which inputs each network have can be seen in table 6.1 below.

All the networks except \hat{m}_{22} have hidden neurons in the interval $[-\pi, \pi]$ with a distance of 0.75 between the units. Since element m_{22} not is dependent on any

variable this network is a bit different and has a constant input, m_{22} . The hidden units here span the interval $[0, 30]$ with a distance of 0.75 between them.

All the \hat{M} networks have 1 input while the \hat{G} and \hat{C} networks all have 2 inputs and are thus much larger due to the number of inputs specifies the network dimension.

For starting the RBFNN sizes are as shown in table 6.1 with the above mentioned intervals and distance.

RBF Neural Networks		
<i>Network</i>	<i>Network Size</i>	<i>Inputs</i>
\hat{m}_{11}	9	q_2
\hat{m}_{12}	9	q_2
\hat{m}_{21}	9	q_2
\hat{m}_{22}	41	1
\hat{c}_{11}	81	q_2, \dot{q}_2
\hat{c}_{12}	81	$q_2, (\dot{q}_2 + \dot{q}_1)$
\hat{c}_{21}	81	q_2, \dot{q}_1
\hat{g}_1	81	q_1, q_2
\hat{g}_2	81	q_1, q_2

Table 6.1: RBF Neural Networks

6.5 Simulations with Correct Model and Unpruned RBFNNs

6.5.1 Correct Model

Here the correct and known matrices is used to control the robot. In this situation there are no disturbances or friction and the is model correct.

In figure 6.4 the trajectory the end effector had with using the correct system matrices can be seen and in 6.5 are the trajectories for the two joints. The sum squared tracking error for the whole period was 0.079779.

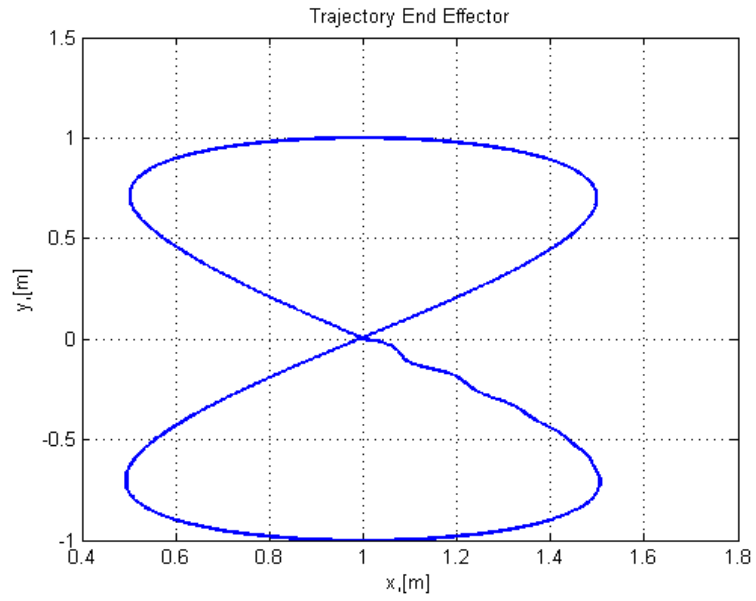


Figure 6.4: Trajectory End Effector Using Correct Model

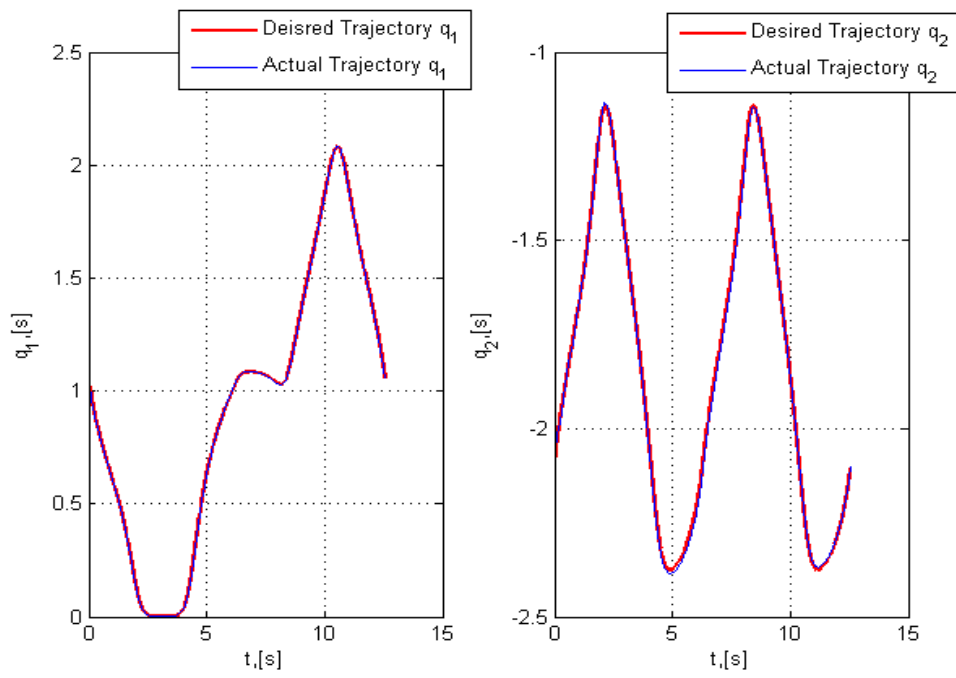


Figure 6.5: Trajectory Joint 1 and 2 Using Correct Model

6.5.2 Unpruned RBFNN

It is also used the networks without pruning them to control the robot. The SSE tracking error was then 0.079723.

For the end effector and for the two joints the resulting trajectory can be seen in figure 6.6 and figure 6.7 respectively. The approximation error for each network along with their sizes is shown in table 6.2.

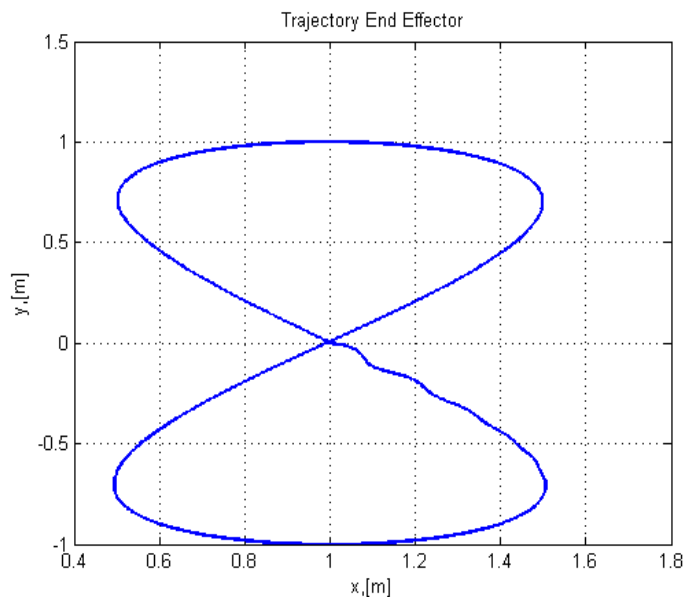


Figure 6.6: Trajectory End Effector Using Unpruned RBFNNs

6.6 Simulations with Online Pruning

In this section the networks used in the inverse kinematic controller are also pruned with weight magnitude pruning, node output pruning and these two methods combined.

Until the simulation has done 1000 iterations (out of a total of 8795 iterations) the networks are not pruned. Also the approximation error has to be smaller than 0.01 over the last 400 iterations before the pruning begins. This is done to avoid starting the pruning too soon. The \hat{G} networks both have worse approximation error so they have here been pruned if the approximation error for the last 400

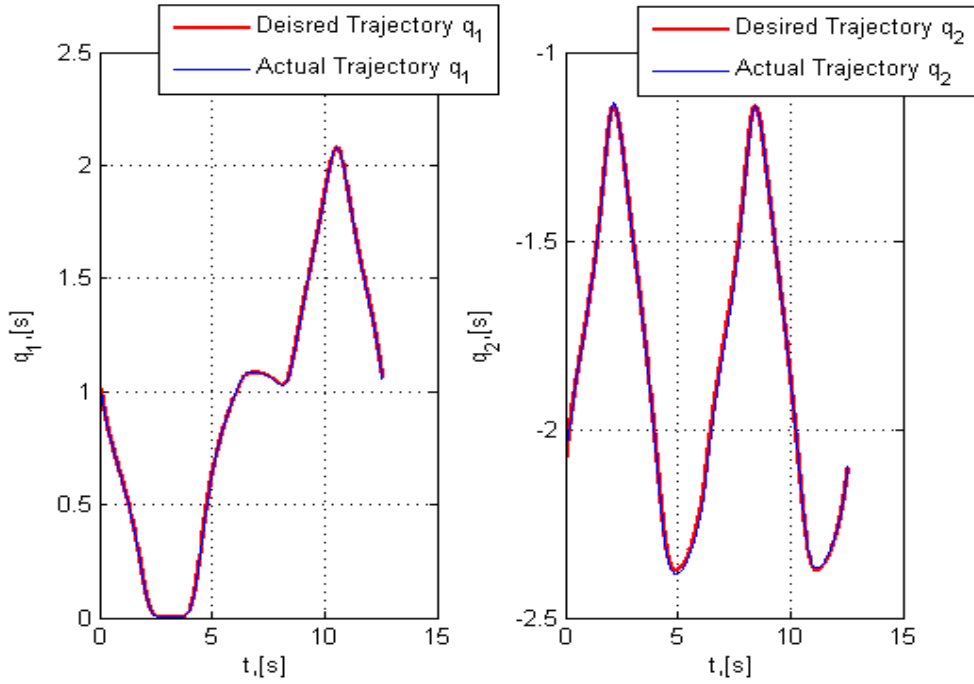


Figure 6.7: Trajectory Joint 1 and 2 Using Unpruned RBFNNs

Unpruned Networks		
<i>Network</i>	<i>Network Size</i>	<i>Error Last 300 Iterations [RMSE]</i>
\hat{m}_{11}	9	0.056283
\hat{m}_{12}	9	0.033363
\hat{m}_{21}	9	0.033363
\hat{m}_{22}	41	0
\hat{c}_{11}	81	0.041565
\hat{c}_{12}	81	0.051779
\hat{c}_{21}	81	0.014998
\hat{g}_1	81	0.1302
\hat{g}_2	81	0.71542
Sum	473	1.076971

Table 6.2: Unpruned Networks

iterations is smaller than 0.03 for \hat{g}_1 and 0.08 for \hat{g}_2 .

Elements m_{12} and m_{21} are identical due to the inertia matrix being symmetric. Two separate networks is still used to approximate the elements to see if the networks end up as the same and if they give different approximation results.

6.6.1 Weight Magnitude Pruned RBFNN

The threshold is set to be 20% of the weight norm and increasing for each time the networks are pruned. Max 5 times can the threshold increase in order to make sure that it does not grow too much. In the simulations done here the networks are never pruned more than 2 times so the growing of the thresholds were never a problem.

This pruning gave the networks in table 6.3. The table shows the final network sizes, the approximation error for each network over the last 300 iterations of the simulation and at what iteration the networks were pruned. Since it here is used growing weight pruning some networks are pruned more than one time. It is then showed at which iteration and how many units that are pruned.

Weight Magnitude Pruned Networks			
<i>Network</i>	<i>Final Size</i>	<i>Error Last 300 Iterations</i> <i>[RMSE]</i>	<i>Pruned at Iteration</i> <i>(Number of Units)</i>
\hat{m}_{11}	3	0.057609	3791(4), 8143(2)
\hat{m}_{12}	3	1.5269	3812(6)
\hat{m}_{21}	3	1.5269	3812(6)
\hat{m}_{22}	1	3.5527×10^{-15}	1001(38), 1002(2)
\hat{c}_{11}	2	0.91704	2648(73), 3728(6)
\hat{c}_{12}	4	0.25724	2453(77)
\hat{c}_{21}	4	0.049003	7106(76), 8028(1)
\hat{g}_1	8	0.099321	7785(73)
\hat{g}_2	8	0.81306	1325(73)
Sum	36	5.247073	-

Table 6.3: Weight Magnitude Pruned Networks

When starting with the networks in table 6.1 and ending up with the networks in table 6.3 the tracking error for the whole simulation was 0.078715. This tracking error is actually better than using the real model and unpruned networks.

In figure 6.8 the trajectories for the two joints together with the desired trajectories are shown. Since the trajectory for the end effector is very similar to the one obtained from the unpruned networks it is here not included a plot of it.

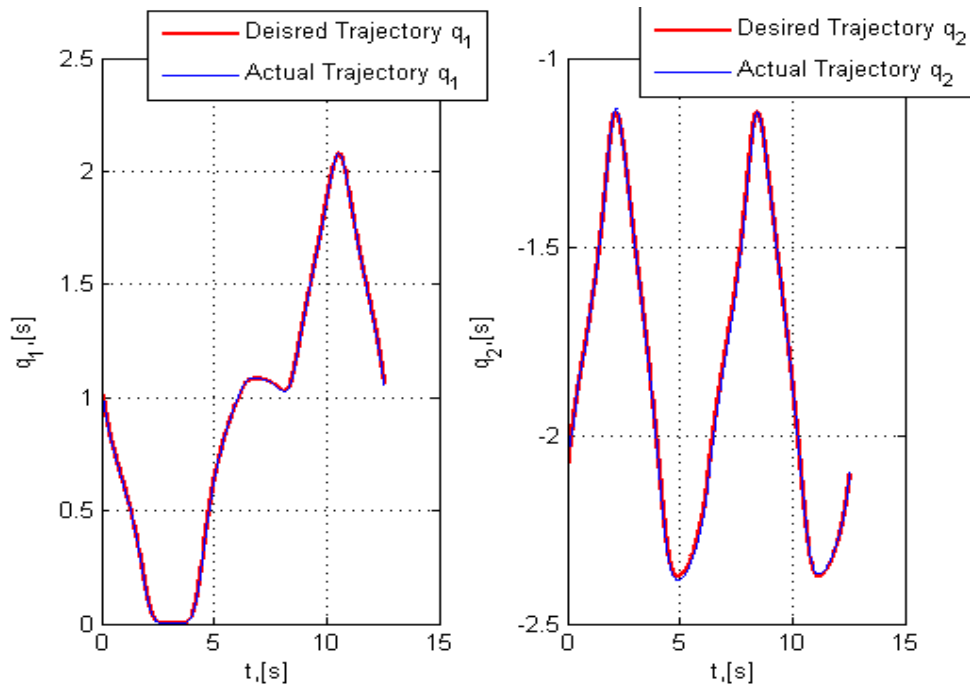


Figure 6.8: Trajectory Joint 1 and 2 Using Weight Pruned RBFNNs

6.6.2 Neuron Output Pruned RBFNN

Here a pruning threshold of 15% of the node output ratio sum is used and the pruned neurons are all removed at the same iteration.

In table 6.4 it is shown the resulting networks when they are pruned with neuron output based pruning. It can be seen what the final number of hidden units was for each network and the root mean squared approximation error for the last 300 iterations of the simulation.

It can in table 6.4 be seen that the total number of nodes is 11 units larger than with only weight magnitude pruning. Also the tracking error is a bit larger. This was here 0.079692 found by the SSE function. However, when looking at the approximation error it can be seen that this is much better than with weight based pruning. The sum of approximation error for 20% weight magnitude pruning was 5.247073 while it here only is 1.3195043.

The trajectory for the joints can be seen in figure 6.9. Even with poorer tracking error the obtained tracking is very accurate as can be seen. It is not included a plot of the end effector trajectory due to this looking exactly the same as the previously shown ones.

Neuron Output Ratio Pruned Networks			
<i>Network</i>	<i>Final Size</i>	<i>Error Last 300 Iterations</i> <i>[RMSE]</i>	<i>Pruned at Iteration</i>
\hat{m}_{11}	3	0.050703	3791
\hat{m}_{12}	3	0.033421	3812
\hat{m}_{21}	3	0.033421	3812
\hat{m}_{22}	3	0	1001
\hat{c}_{11}	10	0.027652	2661
\hat{c}_{12}	8	0.085565	2605
\hat{c}_{21}	9	0.0096223	7113
\hat{g}_1	11	0.1351	7786
\hat{g}_2	7	0.94402	1325
Sum	57	1.3195043	-

Table 6.4: Neuron Output Ratio Pruned Networks

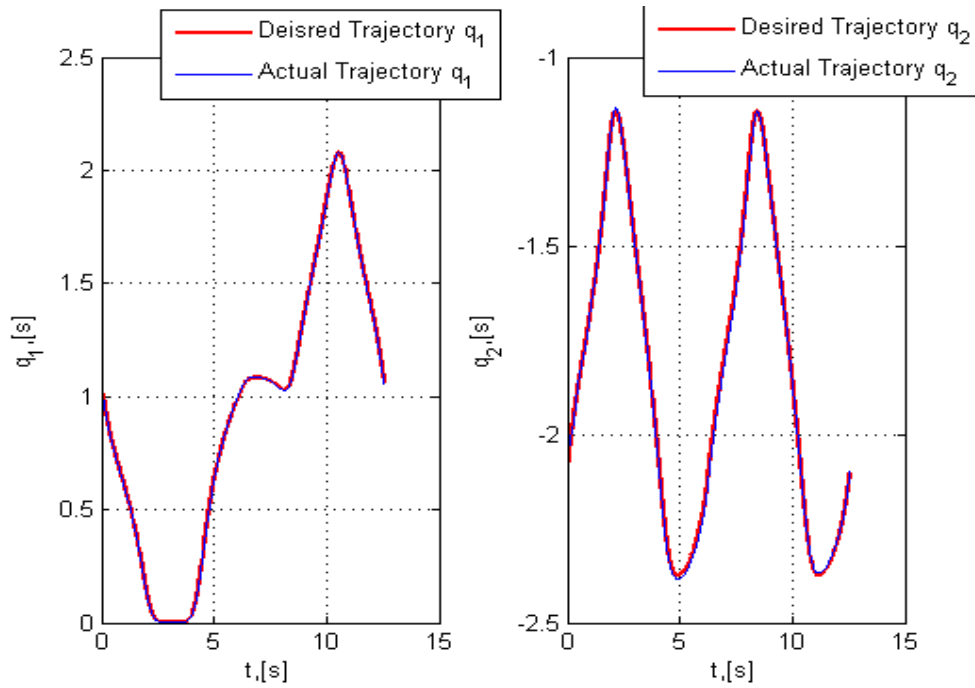


Figure 6.9: Trajectory Joint 1 and 2 Using Node Output Pruned RBFNNs

6.6.3 Mixed Methods Pruned RBFNN

Weight magnitude pruning gave in total fewer hidden units than node output pruning as shown in the two previous sections. The estimations given by the node output pruned networks were however much better. When looking at the networks separately it can be seen that the networks \hat{m}_{11} , \hat{m}_{12} and \hat{m}_{21} had the same sizes for both methods however with much better approximation error when using output ratio based pruning. So now 15% output ratio pruning for these three small networks and 20% weight magnitude pruning for the other larger networks are tried to see if the approximation error could be better for the smallest number of hidden units.

With the above mentioned pruning the total number of hidden nodes is 36 as with only weight magnitude pruning and the RMS approximation error is 2.252727. This is much better than the approximation error from the weight magnitude pruned networks and still not as good as the node output pruned networks. The final networks and approximation errors are shown in table 6.5.

Final tracking sum squared error with these networks was 0.079410 and the joint trajectory is shown in figure 6.10.

Mixed Weight Magnitude and Output Ratio Pruned RBFNNs			
<i>Network</i>	<i>Network Size</i>	<i>Error Last 300 Iterations [RMSE]</i>	<i>Pruned at Iteration (Units Removed)</i>
\hat{m}_{11}	3	0.050711	3791(6)
\hat{m}_{12}	3	0.033426	3812(6)
\hat{m}_{21}	3	0.033426	3812(6)
\hat{m}_{22}	1	3.5527×10^{-15}	1001(38), 1002(2)
\hat{c}_{11}	2	0.91639	2648(73), 3728(6)
\hat{c}_{12}	4	0.25676	2453(77)
\hat{c}_{21}	4	0.04934	7099(76), 8032(1)
\hat{g}_1	8	0.099454	7786(73)
\hat{g}_2	8	0.81322	1325(73)
Sum	36	2.252727	-

Table 6.5: Mixed Weight Magnitude and Output Ratio Pruned RBFNNs

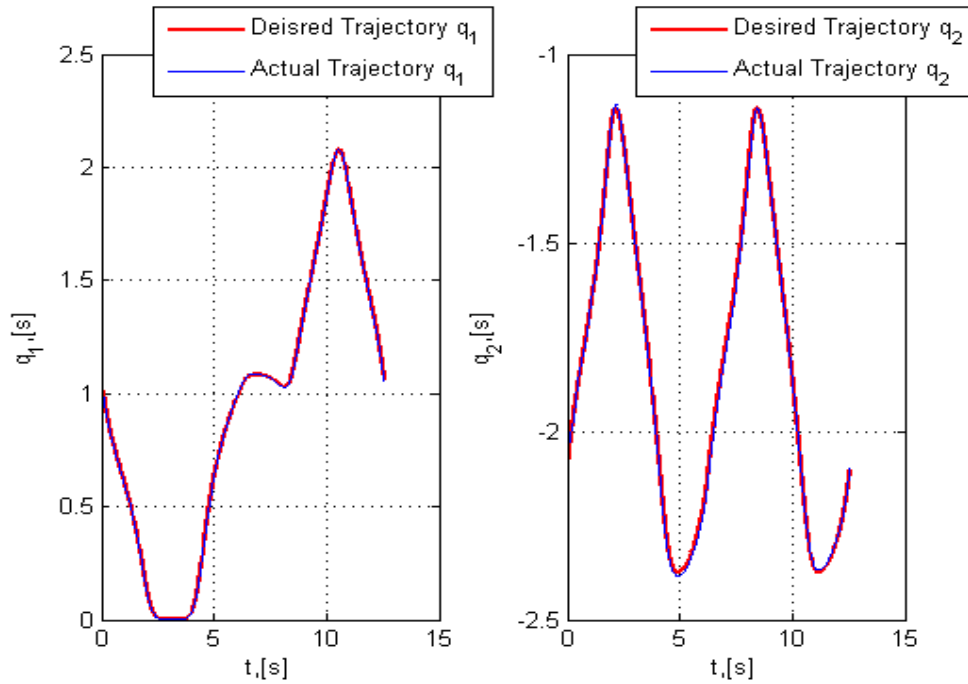


Figure 6.10: Trajectory Joint 1 and 2 Using Mixed Methods Pruned RBFNNs

6.7 Simulations with Friction and a Constant Disturbance

In this section a simple model of friction and some constant disturbance is added to the manipulator and thus the known model is no longer perfect. All the simulations are repeated and the manipulator is controlled with the now incorrect model, unpruned RBFNNs and the different versions of pruned RBF networks. Since the data used to train the networks no longer are entirely correct the networks ability to generalize will be more important.

With friction and constant disturbance the dynamic equation for the robot manipulator can be written as

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) + F(\dot{q}) + d = \tau \quad (6.7.1)$$

where $M(q)$ is the inertia matrix, $C(q, \dot{q})$ is the centrifugal matrix, $G(q)$ is the gravitation matrix, $F(\dot{q})$ is the friction matrix and d is a constant disturbance. q, \dot{q}, \ddot{q} are respectively joint position, joint velocity and joint acceleration.

The friction and disturbance are given as

$$F(\dot{q}) = \begin{bmatrix} 0.7\dot{q}_1 \\ 0.7\dot{q}_2 \end{bmatrix} \quad d = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \quad (6.7.2)$$

It is started the same initial RBF networks as earlier in this chapter for the manipulator without friction and disturbance. These networks are given in table 6.1.

6.7.1 Incorrect Model and Unpruned Networks

Using the incorrect model gave a SSE tracking error of 0.080093 and the trajectory for the end effector can be seen in figure 6.11. This trajectory looked quite similar for all the simulations done in this section and thus this is the only figure of the end effector trajectory included. In figure 6.12 plots of the desired trajectory together with the actual trajectory for both joints are shown.

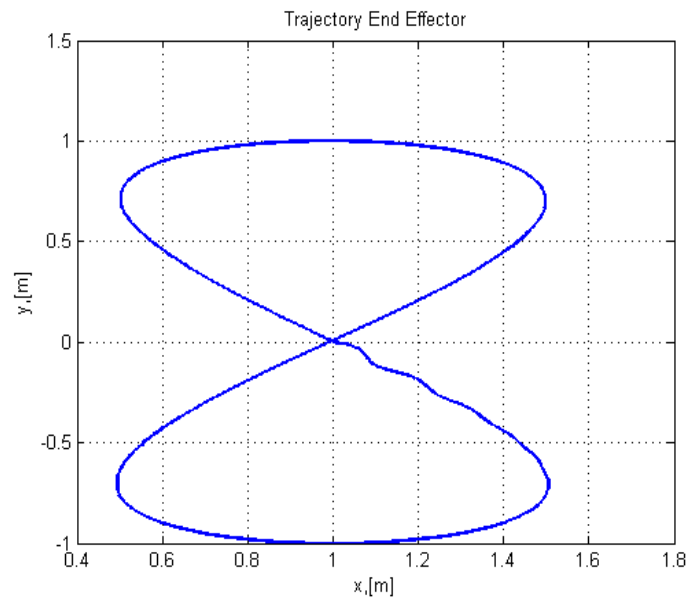


Figure 6.11: Friction and Disturbance - Trajectory End Effector Using Incorrect Model

For the situation of using unpruned RBF networks the final tracking sum squared error was 0.080038. There are no large differences in the plots of the trajectories for using the incorrect model and the unpruned networks and hence

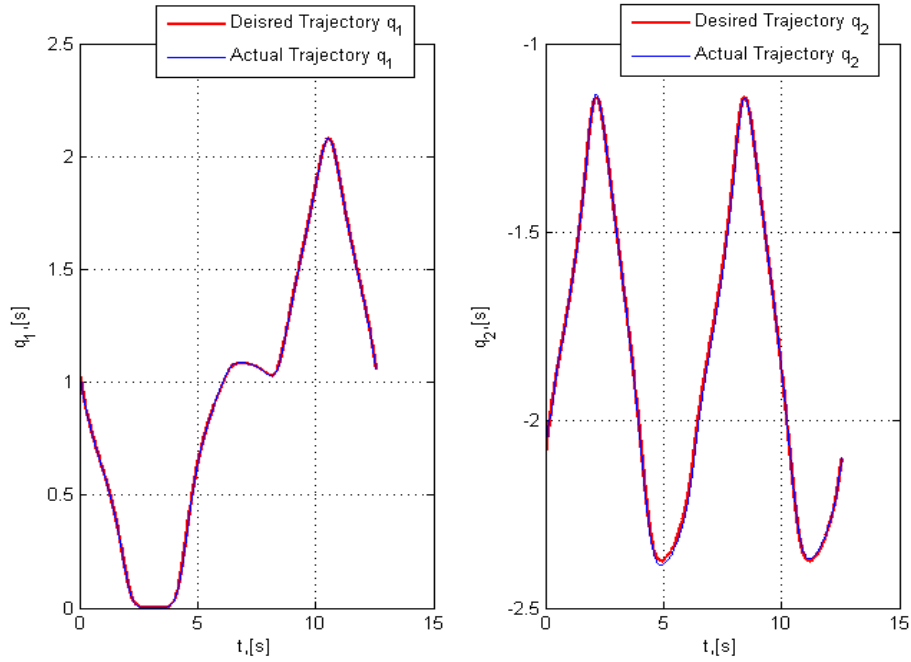


Figure 6.12: Friction and Disturbance - Trajectory Joint 1 and 2 Using Incorrect Model

it is not included a figure of the obtained trajectories for using the unpruned RBFNNs.

6.7.2 Online Pruning

For the online pruning in control of a manipulator with friction and a disturbance criteria for when to begin the pruning are as they were in the situation with no friction and disturbance. These criteria were that the simulation must have completed 1000 iterations and the RMS approximation error for a network has to be smaller than 0.01 over a sliding window of 400 iterations before this network can be pruned. For the networks \hat{g}_1 and \hat{g}_2 the approximation error again has to be smaller than 0.03 and 0.08 respectively before they can be pruned.

The obtained plots of end effector and joint trajectories all look very much alike and thus it is not found any reason to include all of them.

Only Weight Magnitude Pruning

When the networks only are pruned with weight magnitude based pruning threshold of 20% is used as it was when there was no friction and disturbance present. The number of hidden units in the different networks end up as in table 6.3. Final sum squared tracking error for the joints was 0.079035 and in figure 6.13 it is shown a plot of the desired and actual joint trajectories. Even with the minor modeling errors the tracking is very accurate as shown in this figure.

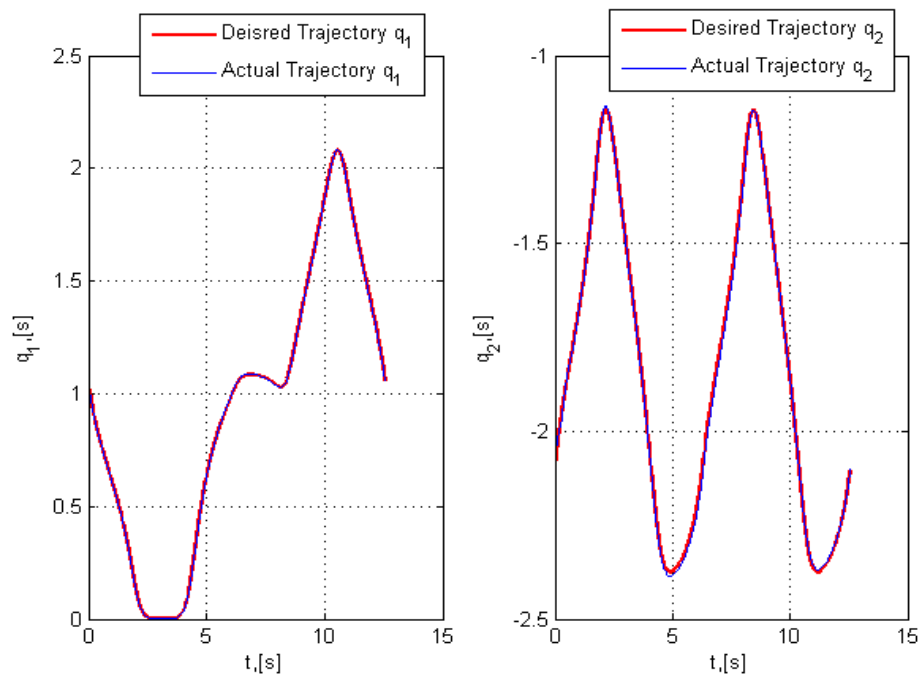


Figure 6.13: Friction and Disturbance - Trajectory Joint 1 and 2 Using Weight Pruned RBFNNs

Only Neuron Output Pruning

Again the threshold is 15% for node output based pruning and the final networks had the same size as those in table 6.4. This gave a tracking sum squared error of 0.08005 which is a bit better than the incorrect model and worse than the weight magnitude pruned networks. The joint trajectories plot is very similar to the one obtained with weight magnitude pruned RBF networks and it is not included here.

Mixed Weight and Neuron Output Pruning

For the networks pruned with either weight based pruning or node output pruning it is done as earlier with 15% node output for the smallest networks \hat{m}_{11} , \hat{m}_{12} and \hat{m}_{21} and 20% weight for the remaining larger ones. This gave the same sizes of the networks as when there were no friction or disturbance in the manipulator and can be seen in table 6.5. The obtained tracking sum squared error for this situation was 0.079727. This is almost as good as when the networks are pruned with only weight based pruning. Again the plot of the trajectories looks like the already shown and is thus not included here.

6.8 Discussion

In table 6.6 all the tracking errors and sum of approximation errors over the last 300 iterations of the simulation for the different ways to implement the system dynamics in the inverse kinematic controller are shown. The situation here is no friction or disturbance in the manipulator and thus the model used to train the networks is correct. It is also included the total number of hidden units for all the final networks.

As can be seen from this table using only weight based pruning for all the networks give the best tracking error while implementation of the correct model actually gives the worst tracking. The best estimations for the networks come from the unpruned networks and second best from the node output pruned networks. The definitely largest approximation error comes from using weight magnitude pruning.

Trajectory Tracking and Network Approximation Errors for Manipulator without Friction and Disturbance			
<i>Method</i>	<i>Sum Hidden Units</i>	<i>Tracking Error [SSE]</i>	<i>Sum Approx. Errors [RMSE]</i>
Correct Model	-	0.079779	-
Unpruned Networks	473	0.079723	1.076971
Weight Based Pruning	36	0.078715	5.247073
Neuron Output Pruning	57	0.079692	1.3195043
Mixed Method Pruning	36	0.079410	2.252727

Table 6.6: Trajectory Tracking and Approximation Errors

For the case when the manipulator has friction and a small disturbance the

tracking errors are as shown in table 6.7. Here the approximation errors are not shown since the training is based on the now incorrect manipulator model. The sizes of the networks are as in table 6.6. Again weight based pruning give the smallest tracking error and the incorrect model has the worst.

Friction and Disturbance - Trajectory Tracking Errors	
<i>Method</i>	<i>Tracking Error [SSE]</i>
Incorrect Model	0.080093
Unpruned Networks	0.080038
Weight Based Pruning	0.079035
Neurong Output Pruning	0.08005
Mixed Method Pruning	0.079727

Table 6.7: Friction and Disturbance - Trajectory Tracking Errors

From simulations done in this chapter it is found that using radial basis function networks to estimate the system dynamics give better tracking than using the actual manipulator model both for the case when this model is incorrect and when the it is perfect. The smallest tracking error is obtained with the weight magnitude pruned networks both for controlling the manipulator with and without friction and a constant disturbance. This is somewhat strange since this pruning method gives the poorest approximations. In the situation where the manipulator model is known and correct then using this should give the best tracking. Here it seems that larger estimation error gives better tracking.

It is also found that using unpruned RBF networks gave better approximation than using the smaller ones obtained with pruning. Out of the different pruned networks it was the node output pruning which gave the networks with best approximation ability.

For the larger networks \hat{G} , \hat{C} and \hat{m}_{22} it was possible to obtain smaller networks with weight magnitude pruning while for the small networks \hat{m}_{11} , \hat{m}_{12} and \hat{m}_{21} both pruning methods gave the same sizes. They did however not remove the exact same neurons and the networks from node output pruning gave much better approximations. Thus could the same small size of only 36 hidden nodes in total for all the networks be kept and the approximation improved by mixing the different pruning methods. It was then used weight based pruning for the larger networks and node output for the smaller networks. The tracking error obtained was larger than the one for using weight magnitude pruning and smaller than using the node output pruning. The approximation error was smaller than for the weight magnitude pruning and larger than for the node output pruning.

In the chapter where the pruning methods were tested on the cross function it was found that first pruning with node output ratio pruning and then weight magnitude pruning in same pruning phase gave the smallest possible network. This was also tested for the networks here and found to not give very good results. Reducing the networks more than to 36 hidden units gave worse errors both in tracking and approximation. Also the pruning time will be much larger as both the pruning algorithms are done. Hence it was found to be unsuited for online pruning in learning manipulator control.

In this chapter the simulation had to complete 1000 iterations before the networks could be pruned in addition to the criterion that the approximation error had to be smaller than 0.01. This was an extra precaution to avoid pruning the networks too early that was not needed in the offline learning with the cross function. To make the approximation error criterion suitable for online learning it was changed to use a moving window of the last 400 iterations for checking the approximation error. Since the \hat{G} networks always had a pretty large estimation error they were implemented to start the pruning if the approximation error was smaller than 0.03 and 0.08 for \hat{g}_1 and \hat{g}_2 respectively. It was with these settings found that the networks were pruned at a bit different iterations but none too early.

For the simulations in this chapter one network was dedicated to element m_{12} and one network to element m_{21} even if they are two equal elements due to the symmetry of the inertia matrix. This was done to see if the networks initially created identically were pruned at the same time and if they gave the same output. It was found that this was the case and thus further on to use a network for each of the elements in the symmetric inertia matrix will not be necessary. When the number of joints increases more several elements will be equal and not needing to use one network for each element would reduce the computational cost significantly.

Here it has been shown that the inverse kinematic controller is able to track a desired trajectory quite well even if the system dynamics are approximated with RBF networks that are pruned.

As with the simulations for the cross function it was clearly neuron output that gave the best approximations compared to the weight magnitude pruning. Weight based pruning did however here give smaller networks and a better tracking error. The fact that the smallest tracking error is obtained with the networks that gave the largest estimation errors is strange and it is difficult to draw any conclusion on which pruning methods that obtained the best result here. Perhaps the networks pruned with the mixed method that were neither best nor worse in any areas would

be the preferred choice in total.

When the pruning methods are implemented in simulation with a larger manipulator they should however all be tested again to see what would give the best results there.

Chapter 7

Online Learning Controller for ABB IRB140

7.1 Introduction

In this chapter the learning inverse kinematic controller with pruned RBF networks is implemented to control the ABB IRB140 robot manipulator in simulations. This manipulator has 6 joints where the last 3 are the wrist. Since the number of elements in the dynamic matrices will increase from 10 as with the 2 dof manipulator to 21 elements for 3 dof it is here started with only control of the first 3 joints.

The different elements are now also depending on more variables, joint angle and joint angular velocity. This gives that the RBFNN have more inputs than they did when there only were 2 joints. Increasing the number of inputs for a network gives more dimensions which again very fast leads to more hidden neurons.

NTNU has a manipulator of the type ABB IRB140 and thus is this exact model used in here. Later it would then be possible to implement and try the learning control scheme on the actual manipulator.

From the results in the previous chapter on learning controller for the 2 dof manipulator it was difficult to conclude on which pruning method that gave the best result. The fact that tracking error was good when approximation error was poor and vice versa is found to be a bit strange. Some of the networks here will also be bigger and testing to see if this has anything to say for the pruning result is necessary. Thus all the pruning methods from last chapter are again tested in this chapter.

The results from the pruned networks are compared to the ones from using the known model, unpruned networks and also growing RBFs known as RANEFs. Simulations for the case of no friction or disturbance in the manipulator and for the case of a simple model of friction and a constant disturbance are done.

Again *Sum Squared Error (SSE)* is used to find the tracking error and *Root Mean Square Error (RMSE)* to find the approximation error.

The simulations are done in Matlab and Simulink with the same RBF networks created by Sigurd Fjordingen. Model of the ABB IRB140 manipulator is found by Stefan Pchelkin.

7.2 ABB IRB140

ABB IRB140 is a small manipulator with 6 revolute joints. The three last joints are however only the wrist so it is possible to control the manipulator in a desired trajectory by only controlling the first three joints. When increasing the number of joints the number of networks will become much larger and also the size of the networks will grow very fast. Hence only the three first dofs are learned by RBF networks in this section.

Image 7.1 shows the ABB IRB140 and is from the data sheet given by ABB on the manipulator B. From the same data sheet it can be read that ABB IRB140 is a fast and compact robot with a reach of 810 mm and can handle a payload of 6 kg. The manipulator may be mounted on the floor, in any angle on the wall or it can be mounted inverted. Some of the things ABB IRB140 is suited for are arc welding, assembly, cleaning/spraying, machine tending, material handling packing and deburring.

7.3 Controller and Desired Trajectory

Again the controller is the inverse kinematic controller from [18] and is restated here in the learning form

$$\tau = \hat{M}(q)(\ddot{q}_d + K_d(\dot{q}_d - \dot{q}) + K_p(q_d - q)) + \hat{C}(q, \dot{q})\dot{q}_d + \hat{G}(q) \quad (7.3.1)$$

The gains are now

$$K_p = \begin{bmatrix} 350 & 0 & 0 \\ 0 & 450 & 0 \\ 0 & 0 & 450 \end{bmatrix} \quad (7.3.2)$$



Figure 7.1: ABB IRB140

and

$$K_d = \begin{bmatrix} 2.5 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 2.5 \end{bmatrix} \quad (7.3.3)$$

For each joint the desired trajectory is specified as

$$q_{1d}(t) = 0.5\sin(t) \quad (7.3.4)$$

$$q_{2d}(t) = 0.7\sin(t) \quad (7.3.5)$$

$$q_{3d}(t) = 0.3\sin(t) \quad (7.3.6)$$

In figure 7.2 below the desired trajectory for the joints can be seen.

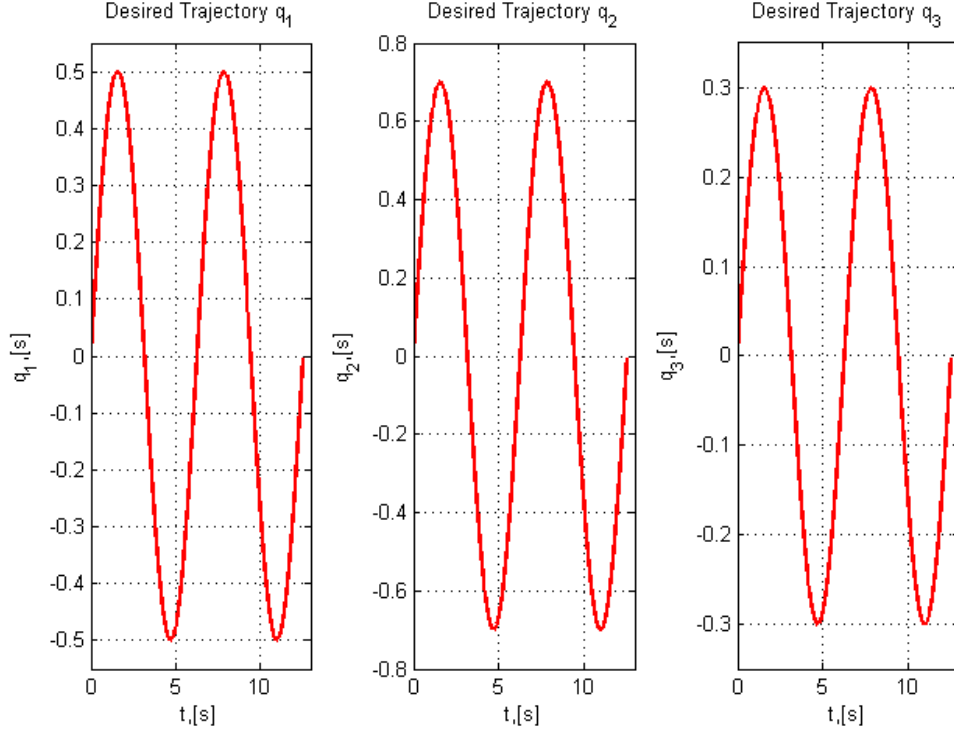


Figure 7.2: Desired Trajectory for ABB IRB140

7.4 RBF networks

Again the notation from chapter 2.5 to find the estimated dynamics of the manipulator for implementation in the inverse kinematic controller is used.

$$\hat{M} = W_M^T \bullet A_M(x) = \begin{bmatrix} \mathbf{w}_{m11}^T \mathbf{a}_{m11}(x) & \mathbf{w}_{m12}^T \mathbf{a}_{m12}(x) & \mathbf{w}_{m13}^T \mathbf{a}_{m13}(x) \\ \mathbf{w}_{m21}^T \mathbf{a}_{m21}(x) & \mathbf{w}_{m22}^T \mathbf{a}_{m22}(x) & \mathbf{w}_{m23}^T \mathbf{a}_{m23}(x) \\ \mathbf{w}_{m31}^T \mathbf{a}_{m31}(x) & \mathbf{w}_{m32}^T \mathbf{a}_{m32}(x) & \mathbf{w}_{m33}^T \mathbf{a}_{m33}(x) \end{bmatrix} \quad (7.4.1)$$

Finding the approximated C matrix follows the same pattern

$$\hat{C} = W_C^T \bullet A_C(x) = \begin{bmatrix} \mathbf{w}_{c11}^T \mathbf{a}_{c11}(x) & \mathbf{w}_{c12}^T \mathbf{a}_{c12}(x) & \mathbf{w}_{c13}^T \mathbf{a}_{c13}(x) \\ \mathbf{w}_{c21}^T \mathbf{a}_{c21}(x) & \mathbf{w}_{c22}^T \mathbf{a}_{c22}(x) & \mathbf{w}_{c23}^T \mathbf{a}_{c23}(x) \\ \mathbf{w}_{c31}^T \mathbf{a}_{c31}(x) & \mathbf{w}_{c32}^T \mathbf{a}_{c32}(x) & \mathbf{w}_{c33}^T \mathbf{a}_{c33}(x) \end{bmatrix} \quad (7.4.2)$$

and so does also the estimation of G

$$\hat{G} = W_G^T \bullet A_G(x) = \begin{bmatrix} \mathbf{w}_{g1}^T \mathbf{a}_{g1}(x) \\ \mathbf{w}_{g2}^T \mathbf{a}_{g2}(x) \\ \mathbf{w}_{g3}^T \mathbf{a}_{g3}(x) \end{bmatrix} \quad (7.4.3)$$

Since the inertia matrix is symmetric only needed 6 networks instead of 9 are needed to learn all the elements in $M(q)$. The similar elements are

$$m_{12} = m_{21} \quad (7.4.4)$$

$$m_{13} = m_{31} \quad (7.4.5)$$

$$m_{23} = m_{32} \quad (7.4.6)$$

Also c_{33} and g_1 are known to be zero and thus not learned by any network. This gives a total of 16 networks.

All the networks are spanned in the interval $[-\pi, \pi]$ with a distance of 0.75 between the nodes. In table 7.1 the initial networks and the inputs to the networks can be seen. Some of the joint velocities are added up and fed to the network as one input in order to reduce the number of inputs and then reduce the size of the networks.

7.5 Simulations with Correct Model and Unpruned RBFNN

In this section the known and correct model is first used in the controller and then unpruned RBF networks. Simulation time is $6 * \pi$ so the manipulator finishes 3 whole sinus waves. This gives a simulation time of approximately 18.85 seconds. The time the simulation actually uses is in this and the next sections taken in order to give a picture of the computational cost of each method.

It is also done the simulations with 20π simulation seconds which is approximately 1.03 minutes instead of 6π . This would show how the networks develop over a longer period of time. Also here it is found the actual time the simulation takes.

Shown in the tables in this chapter is the approximation error for the last 300 iterations of the whole simulation in order to show to approximation error when the networks are trained and also in the next section pruned. This is the same as what is done earlier in this report for the simulations with the 2 dof manipulator.

ABB IRB140 RBF Networks		
<i>Network</i>	<i>Network Size</i>	<i>Inputs</i>
\hat{m}_{11}	81	q_2, q_3
\hat{m}_{12}	81	q_2, q_3
\hat{m}_{13}	81	q_2, q_3
\hat{m}_{22}	9	q_3
\hat{m}_{23}	9	q_3
\hat{m}_{33}	9	1
\hat{c}_{11}	729	$q_2, q_3, (\dot{q}_2 + \dot{q}_3)$
\hat{c}_{12}	729	$q_2, q_3, (\dot{q}_2 + \dot{q}_2 + \dot{q}_3)$
\hat{c}_{13}	729	$q_2, q_3, (\dot{q}_2 + \dot{q}_2 + \dot{q}_3)$
\hat{c}_{21}	729	\dot{q}_1, q_2, q_3
\hat{c}_{22}	81	q_3, \dot{q}_3
\hat{c}_{23}	81	$q_3, (\dot{q}_2 + \dot{q}_3)$
\hat{c}_{31}	81	$(\dot{q}_1 + q_2), q_3$
\hat{c}_{32}	81	\dot{q}_2, q_3
\hat{g}_1	81	q_2, q_3
\hat{g}_2	81	q_2, q_3
Sum	3672	-

Table 7.1: ABB IRB140 RBF Networks

7.5.1 Correct Model

When the model of the manipulator is known and completely correct the inverse kinematic controller with this model gives a tracking error of 0.031897 for the whole simulation period when using the sum squared error function (SSE). The desired trajectory together with the actual trajectory for the case when the controller uses the perfect model can be seen in figure 7.3. It can be seen that the tracking is very accurate.

Using the correct model gives a fast simulation and it only takes 25 seconds to finish the 18.85 simulation seconds. When the simulation time is increased to 20π seconds it takes 1.44 minutes for the simulation to complete. The tracking error for the whole 1.03 simulation minutes was 0.031897.

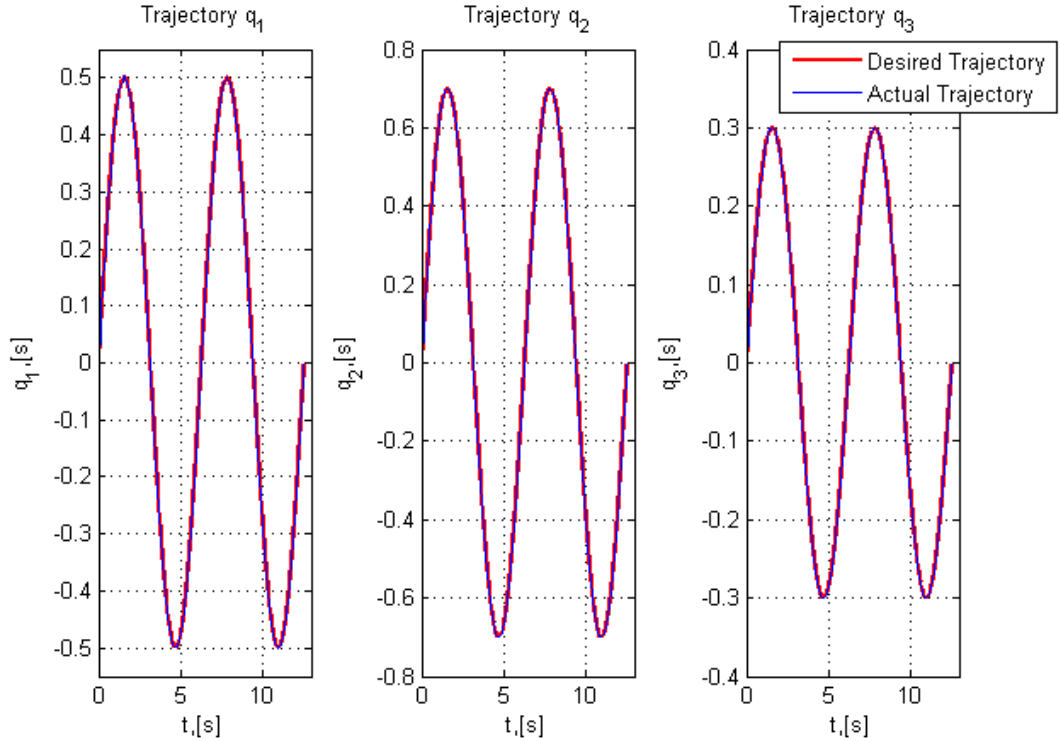


Figure 7.3: Trajectory all 3 Joints Using Correct Model

7.5.2 Unpruned Radial Basis Function Neural Networks

For the case of using not pruned RBF networks the sum squared tracking error was 0.032 for the whole 6π simulation time. This is a bit worse than using the perfect model as would be expected. It is also as expected that this controller makes the simulation slower and it takes 4.49 minutes to complete the 18.85 simulation seconds.

The total approximation error for all the 16 networks was 0.26303 as can be seen in table 7.2. Here the approximation errors for each network over the last 300 iterations of the simulation are also shown.

In figure 7.4 the desired trajectory and the actual trajectory for this situation are visible. It can here be seen that using the initial networks give a very good tracking result even if the total tracking error is slightly higher than for using the correct model of the manipulator.

When the simulation is increased to 20π it takes 25.30 minutes to complete the simulation of approximately 1.03 minutes. The sum squared tracking error for the whole long simulation time was found to be 0.032015.

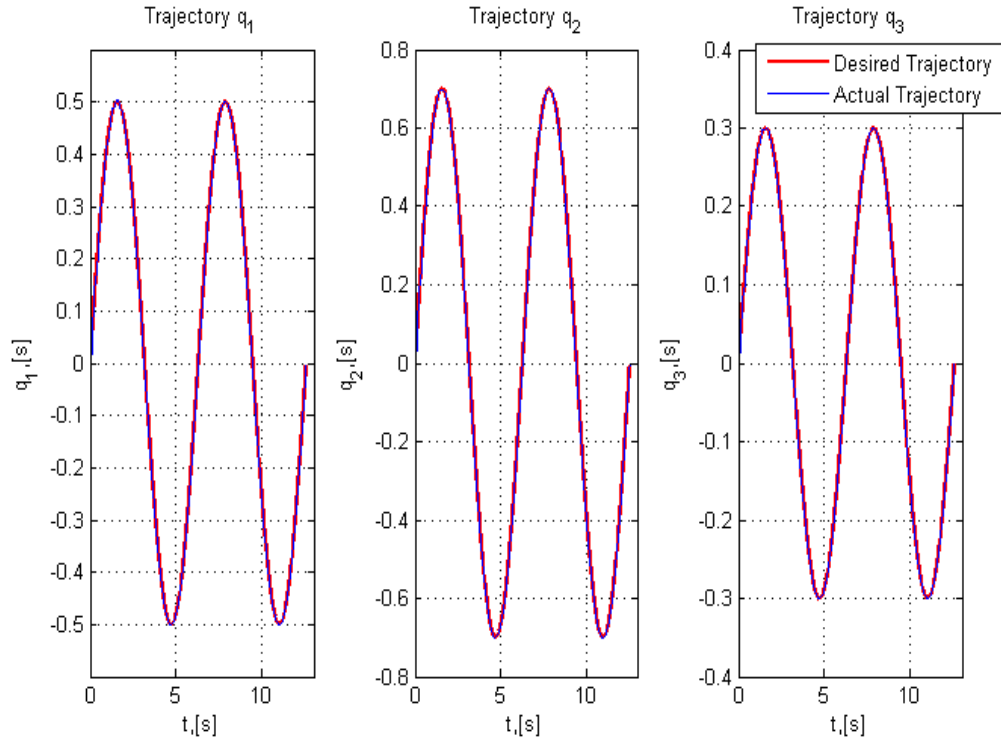


Figure 7.4: Trajectory all 3 Joints Using Unpruned RBFNN

7.6 Simulations with Online Pruning

Here the networks from table 7.1 is implemented and pruned during simulation with weight magnitude pruning, node output pruning or a mixed version of the two pruning methods just mentioned.

As with the simulations for the 2 dof the rms estimation error for the last 400 iterations has to be smaller than 0.01 before a network can be pruned. An exception to this is the network \hat{g}_2 which can be pruned when the approximation error is less than 0.05 for the last 400 iterations. Since the desired trajectory here is a sinus wave the manipulator must have completed one half period of the first sinus wave before the networks can be pruned. This is the same as π simulation seconds which is found to be approximately 2200 iterations. Thus the pruning for a network may only begin when the simulation has completed π simulation seconds and the network has an estimation error smaller than 0.01 over the last

Unpruned Networks		
<i>Network</i>	<i>Network Size</i>	<i>Approx. Error Last 300 Iterations [RMSE]</i>
\hat{m}_{11}	81	0.0066688
\hat{m}_{12}	81	0.00058727
\hat{m}_{13}	81	0.0006894
\hat{m}_{22}	9	0.0041514
\hat{m}_{23}	9	0.001951
\hat{m}_{33}	9	4.4409×10^{-16}
\hat{c}_{11}	729	0.0018134
\hat{c}_{12}	729	0.0021467
\hat{c}_{13}	729	0.0013094
\hat{c}_{21}	729	0.0031408
\hat{c}_{22}	81	0.0013634
\hat{c}_{23}	81	0.00040688
\hat{c}_{31}	81	0.0041832
\hat{c}_{32}	81	0.00058631
\hat{g}_1	81	0.23208
\hat{g}_2	81	0.0019531
Sum	3672	0.26303

Table 7.2: Unpruned Networks

400 iterations.

In this section a simulation time of first 6π which is approximately 18.85 seconds and a simulation time of 20π which approximately is 1.03 minutes are used. It is found the sum squared tracking error for the whole period and the approximation error for the last 300 iterations of the simulation for both simulation lengths. This makes it possible to see how the pruned and trained networks perform when the training length is increased.

7.6.1 Weight Magnitude Pruned RBFNN

In this section the results from when the networks are pruned with 5% weight magnitude pruning are given.

The threshold specified here is the starting threshold and for each time the networks are pruned this threshold grows as it did for the 2 dof weight based pruning. In the second pruning the threshold is twice the threshold it was in the first pruned.

ing while the threshold in the third pruning is three times the starting threshold, etc. It can only grow 5 times and after that the threshold stays the same. The networks that initially were quite large, 729 hidden units, are here pruned over many periods. Network \hat{c}_{21} is pruned most times and this is 9 periods. The nodes are however mainly removed during the first pruning and only a few are pruned in the later repetitions.

As an example at what iterations \hat{c}_{21} is pruned and how many neurons that are removed at the different iteration are now mentioned. At iteration 2201 706 nodes is removed, 10 at iteration 2202, 2 at the next iteration 2203, then one unit is removed at the iterations 2757, 2845, 4009, 4470 and 4764 before the last 3 neurons are pruned at iteration 5150.

The reason for that there are neurons pruned several times after the threshold has stopped growing is that the threshold is specified as a percent of the weights norm. While it for the case with the cross function was found that the norm of the weights converged in this situation it is found that the norm does not converge.

In fact, for some of the networks the norm of the weights increases with time. Thus it is found necessary to end the pruning since the networks may be too much pruned if the simulation time is increased. The pruning lasts for 2π simulation seconds which is one period of the sinus wave.

In table 7.3 the resulting networks after the learning has been implemented with 5% weight based pruning are shown and the simulation time was 6π . Using a starting threshold of 5% gave a total of 74 hidden units for all the networks and the total approximation error summed together for all the networks was 0.29037. With these networks in the inverse kinematic controller the tracking sum squared error was 0.03201 for the whole simulation period. The actual time the simulation took was 4.10 minutes.

In figure 7.5 the trajectories for each joint can be seen. Here the desired trajectory is plotted together with the actual trajectory. As can be seen from this figure the controller implemented with the weight magnitude pruned networks give a good tracking result. The tracking is as shown in this plot accurate in the beginning before the networks are pruned and also after the RBFNNs are pruned several times.

The tracking error was 0.032053 for the whole 20π simulation time which took 23.30 minutes to complete. This is a bit faster than using the unpruned networks. For the last 300 iterations the approximation error was 0.28429 which is smaller

Weight Magnitude Pruned Networks		
<i>Network</i>	<i>Final Size</i>	<i>Error Last 300 Iterations [RMSE]</i>
\hat{m}_{11}	6	0.0067127
\hat{m}_{12}	6	0.0005721
\hat{m}_{13}	2	0.001048
\hat{m}_{22}	3	0.004117
\hat{m}_{23}	3	0.001939
\hat{m}_{33}	2	4.4409×10^{-16}
\hat{c}_{11}	3	0.0083142
\hat{c}_{12}	13	0.0023642
\hat{c}_{13}	8	0.0072465
\hat{c}_{21}	3	0.0072956
\hat{c}_{22}	2	0.0061949
\hat{c}_{23}	3	0.0006785
\hat{c}_{31}	2	0.0069534
\hat{c}_{32}	2	0.0014019
\hat{g}_2	10	0.23334
\hat{g}_3	6	0.0021934
Sum	74	0.29037

Table 7.3: Weight Magnitude Pruned Networks

than the estimation error at the end of the shorter simulation time.

7.6.2 Neuron Output Pruned RBFNN

For neuron output pruning it is found to be necessary for the manipulator to finish one whole period of the sinus wave trajectory before pruning the networks. This gives that the networks are not pruned until the simulation has completed 2π of the simulation time which corresponds to approximately 4400 iterations in simulink.

In this section the results from pruning with neuron output based pruning with 30% thresholds are given. When the simulation time is 6π the final networks end up with the sizes and approximation errors over the last 300 iterations as shown in table 7.4.

Pruning with neuron output ratio has for all the simulations done so far given better approximation ability for the final networks than using weight magnitude

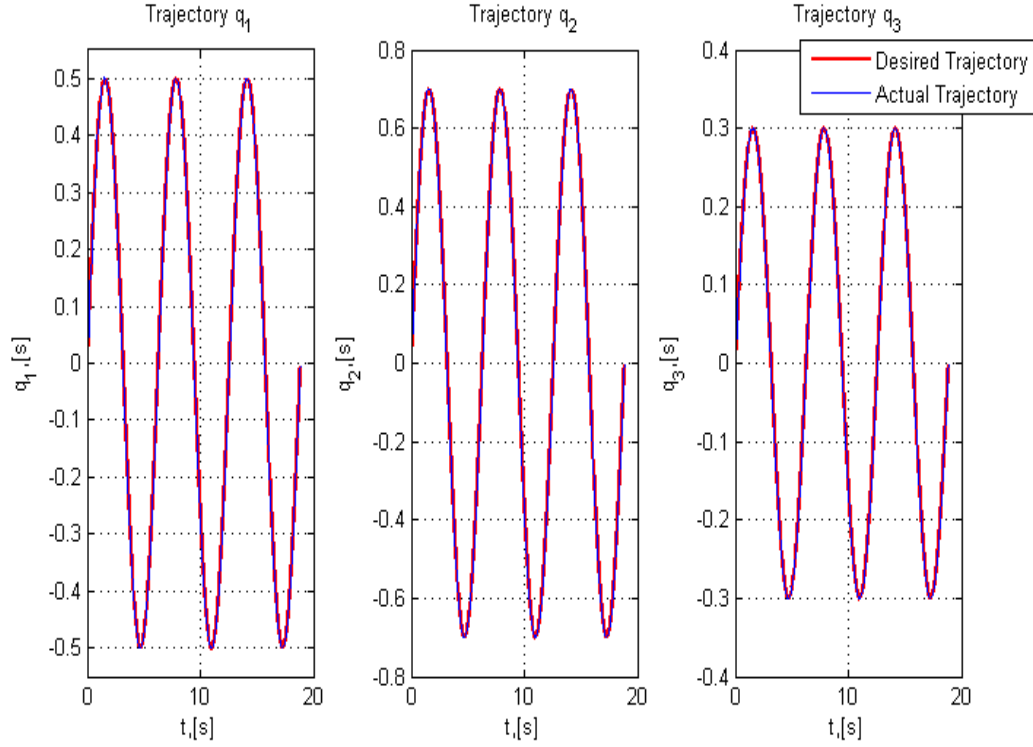


Figure 7.5: Trajectory all 3 Joints Using Weight Pruned RBFNN

pruning. This is found to be the case also in the simulations done here, however with bigger networks than for weight based pruning. The total number of hidden neurons here was 126 and the sum of all the root mean squared approximation errors was 0.28021. Since the estimations are a bit better smaller tracking error than for weight magnitude pruning is obtained and the final tracking error for the whole simulation was 0.032002. It took 4.20 minutes to complete the 18.85 simulation seconds something which is a bit more than for the weight based pruning situation.

The threshold for neuron output based pruning is not growing and the networks are usually pruned 1-3 times with almost all of the nodes removed at the first pruning. This however is not the case for the larger \hat{C} networks with initially 729 hidden units. Also here most of the units are removed at the first pruning and then several pruning phases where additionally one unit at the time is removed follow.

Network \hat{c}_{13} is pruned over 7 periods and this is the network pruned most times. Here first 701 units are removed at iteration 4401 and then one more at the iterations 4616, 4722, 4818, 4926, 7134 and 7178.

The smaller networks are pruned between iteration 4401 and 5700.

Figure 7.6 below shows the joint trajectories obtained when the RBF networks were pruned with 30% threshold together with the desired trajectory. As can be seen from this plot the obtained tracking is very accurate.

Neuron Output Pruned Networks		
<i>Network</i>	<i>Final Size</i>	<i>Error Last 300 Iterations [RMSE]</i>
\hat{m}_{11}	5	0.0061364
\hat{m}_{12}	5	0.00050466
\hat{m}_{13}	5	0.00075247
\hat{m}_{22}	2	0.0065574
\hat{m}_{23}	2	0.0028382
\hat{m}_{33}	2	0
\hat{c}_{11}	17	0.0016613
\hat{c}_{12}	22	0.0037684
\hat{c}_{13}	22	0.00073191
\hat{c}_{21}	11	0.00084954
\hat{c}_{22}	4	0.0012086
\hat{c}_{23}	7	0.0010091
\hat{c}_{31}	6	0.0043594
\hat{c}_{32}	6	0.0011061
\hat{g}_2	5	0.23923
\hat{g}_3	5	0.0095059
Sum	126	0.28021

Table 7.4: Neuron Output Pruned Networks

The simulation time is also here increased to 20π and it took 23.25 minutes to complete the simulation. This is 5 seconds less than for the case with weight based pruned networks. After this simulation it is found that the final sum squared tracking error for the whole simulation was 0.0320025 for all joints. The root mean squared approximation error for the last 300 iterations is 0.2657 which is an improvement from the estimation error at the end of the simulation of 6π .

7.6.3 Mixed Methods Pruned RBFNN

Here it is tried to use weight magnitude pruning with different threshold for the \hat{M} and \hat{G} networks while it is used node output pruning for the \hat{C} networks. This

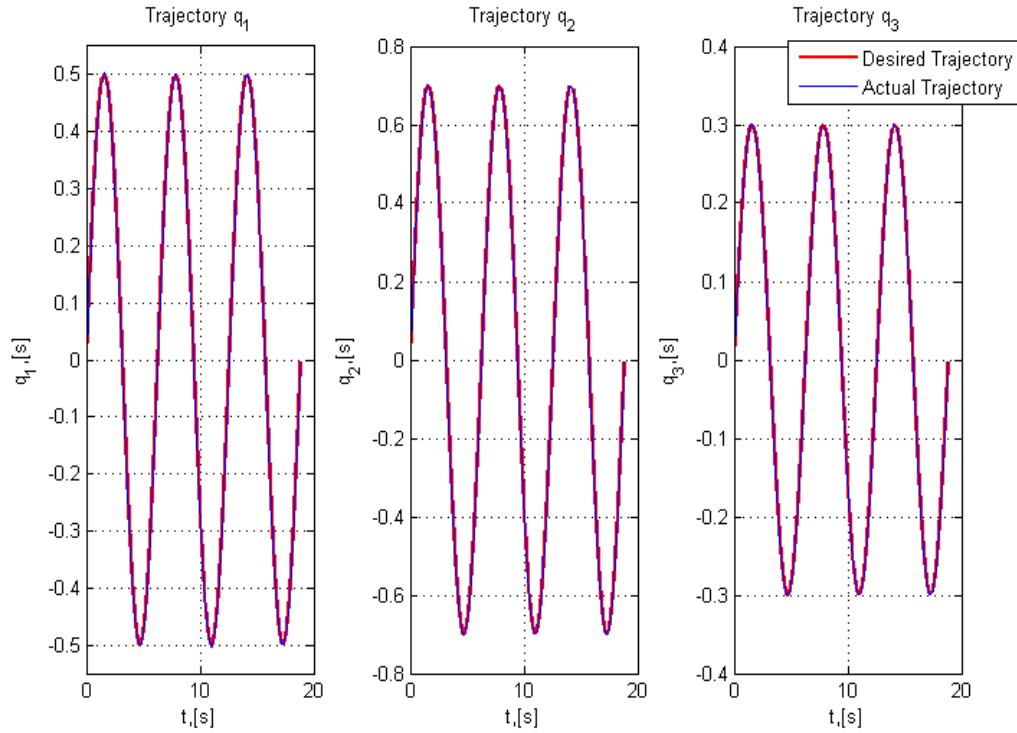


Figure 7.6: Trajectory all 3 Joints Using Neuron Output Pruned RBFNN

is done in order to obtain the networks with best possible approximation ability that still are quite small.

It has been used 7% growing weight threshold to prune all the \hat{M} networks, 30% neuron output pruning for the \hat{C} networks and 5% growing threshold for weight based pruning of the two \hat{G} networks. This gave 129 hidden units in total for all the RBFNNs as can be seen in table 7.7 below. The tracking error was 0.032001 for the whole simulation time of 18.85 seconds and the approximation error in total for all the networks over the last 300 iterations was 0.26314. These errors both are smaller than for the other two pruning methods. It took 4.20 minutes to complete the whole simulation which is the same as for node output pruning alone.

A plot of the trajectory for each joint along with the desired trajectory can be seen in figure 7.7. As with the other controllers it was also here obtained an accurate tracking.

After a simulation time of 20π second the networks gave an approximation rms

Mixed Methods Pruned Networks		
<i>Network</i>	<i>Network Size</i>	<i>Approx. Error Last 300 Iterations [RMSE]</i>
\hat{m}_{11}	4	0.0054748
\hat{m}_{12}	4	0.00034923
\hat{m}_{13}	2	0.0010479
\hat{m}_{22}	3	0.0041168
\hat{m}_{23}	3	0.0019389
\hat{m}_{33}	2	0
\hat{c}_{11}	17	0.0016601
\hat{c}_{12}	22	0.0037649
\hat{c}_{13}	22	0.00073047
\hat{c}_{21}	11	0.00084958
\hat{c}_{22}	4	0.0012063
\hat{c}_{23}	7	0.0010054
\hat{c}_{31}	6	0.0043596
\hat{c}_{32}	6	0.0011059
\hat{g}_1	10	0.23334
\hat{g}_2	6	0.002195
Sum	129	0.26314

Table 7.5: Mixed Methods Pruned Networks

error of 0.25074 over the last 300 iterations which is better than at the end of the shorter simulation and also better than the networks pruned with only weight based pruning or only node output pruning. The tracking error for the whole period was here 0.032018 and it took 23.40 minutes to finish this simulation. This is a bit longer than for either weight magnitude pruning or node output pruning alone.

7.7 Simulations with Resource Allocation Network EKF

In this section growing RBF networks are implemented in the controller instead of creating initially large networks and pruning them. The growing networks used are *Resource Allocation Networks Extended Kalman Filter (RANEKF)* which are briefly described in chapter 2 in this report.

For all the RANEKFs used to learn the dynamics of the manipulator there are only one output as with the RBFNNs and the number of inputs are also the same

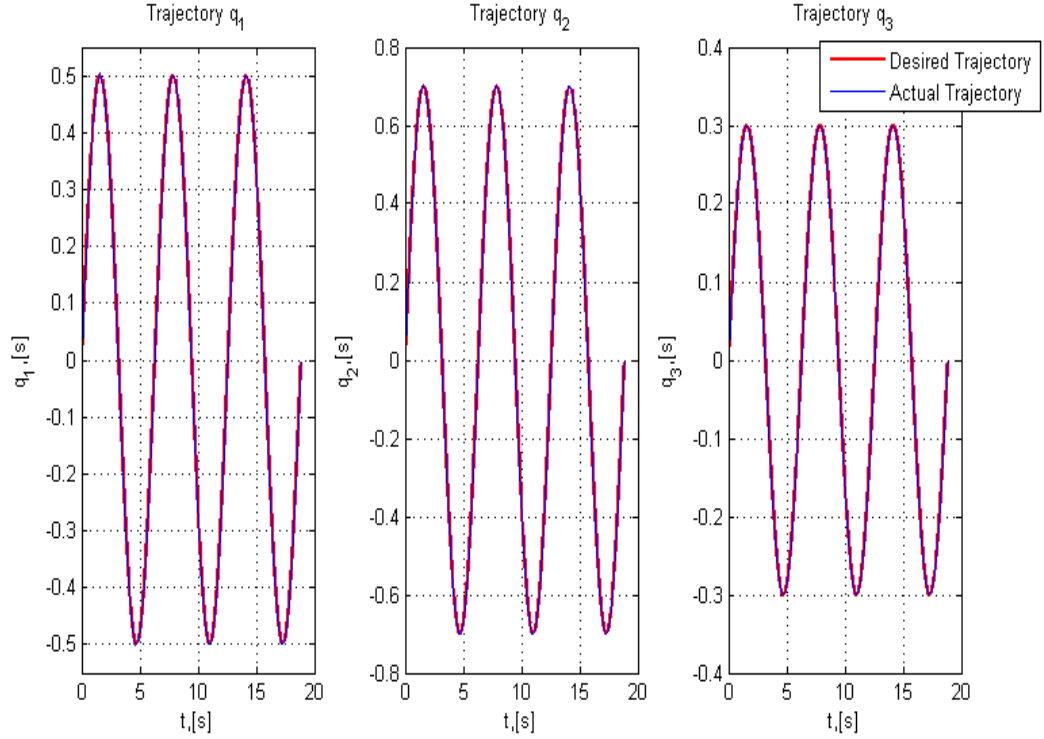


Figure 7.7: Trajectory all 3 Joints Using Mixed Methods Pruned RBFNN

as with the corresponding RBFNNs. The networks are initially completely empty in the hidden layer. To specify whether a new unit should be added to the network or not the following thresholds have been used

$$\epsilon_n = 0.1 \quad (7.7.1)$$

$$e_{min} = 0.1 \quad (7.7.2)$$

$$e'_{min} = 0.1 \quad (7.7.3)$$

where ϵ_n is the size that the shortest distance from a new input to a neuron has to be larger than, e_{min} is the number that the approximation error for the network to the new input has to be larger than and e'_{min} is the value that the root mean square approximation error over the last M inputs has to be larger than. If all these thresholds are met for a new input there is added a new hidden unit.

After 18.85 simulation seconds the final network had 96 hidden units and the approximation error for all the networks was only 0.024542 over the last 300 iterations. This is a significantly improvement from the pruned networks. It took however 7.20 minutes to run the whole simulation something which is quite a

lot of time. The tracking error for the three joints when these RANEKF's were implemented was 0.032094 found with sum squared error for the whole simulation.

All the resulting networks and their estimation errors can be seen in table 7.6 and the joint trajectory plotted with the desired trajectory can be seen in figure 7.8. The obtained tracking is again very accurate.

Resource Allocation Networks EKF		
<i>Network</i>	<i>Final Size</i>	<i>Approx. Error Last 300 Iterations [RMSE]</i>
\hat{m}_{11}	2	0.00081226
\hat{m}_{12}	3	0.00080336
\hat{m}_{13}	2	0.0013159
\hat{m}_{22}	1	0.00024869
\hat{m}_{23}	1	0.00011871
\hat{m}_{33}	1	0
\hat{c}_{11}	13	0.00072606
\hat{c}_{12}	19	0.0020651
\hat{c}_{13}	15	0.00017359
\hat{c}_{21}	4	0.0043685
\hat{c}_{22}	3	0.00053882
\hat{c}_{23}	7	0.00013039
\hat{c}_{31}	3	0.0020125
\hat{c}_{32}	6	0.00021268
\hat{g}_1	14	0.009901
\hat{g}_2	2	0.0011144
Sum	96	0.024542

Table 7.6: Resource Allocation Networks EKF

When the RANEKF networks were implemented in the controller it took 27.05 minutes to complete the simulation of 20π . The networks had then grown to have a total of 147 hidden units. This is many more hidden units than after the 6π simulation and the approximation error over the last 300 iterations was now increased to be 0.058592. Still a very small error but it is more than twice as much as after the shorter simulation. The tracking error for all three joints over the whole period was 0.032095.

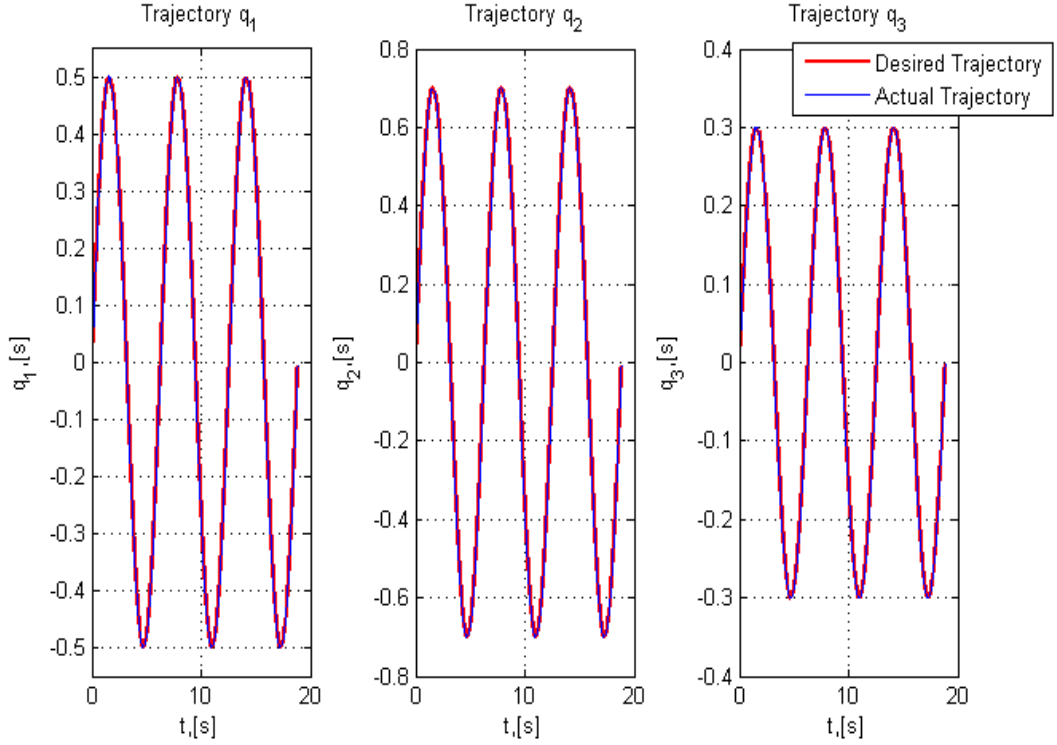


Figure 7.8: Trajectory all 3 Joints Using RANEKFs

7.8 Simulations with Friction and a Constant Disturbance

Now a simple model of friction and some constant disturbance is added to the manipulator which make the known model incorrect. All the simulations done earlier in this chapter are now repeated for the different methods to implement the system dynamics in the inverse kinematic controller. The different methods are first the incorrect model, then unpruned RBF networks, pruned RBF networks with either weight magnitude pruning, neuron output based pruning or a mixed version of these two pruning methods and at last the growing resource allocation network extended kalman filter (RANEKF).

As in the previous sections the initially networks which can be found in table 7.1 are used. The simulation time is again set to 6π seconds so the trajectory has completed 3 whole sinus waves.

With friction and constant disturbance can the dynamic equation for the robot

manipulator be written as it was in the previous chapter and repeated here

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) + F(\dot{q}) + d = \tau \quad (7.8.1)$$

where $M(q)$ is the inertia matrix, $C(q, \dot{q})$ is the centrifugal matrix, $G(q)$ is the gravitation matrix, $F(\dot{q})$ is the friction matrix and d is a constant disturbance. q, \dot{q}, \ddot{q} are respectively joint position, joint velocity and joint acceleration.

Now the friction and disturbance are implemented as

$$F(\dot{q}) = \begin{bmatrix} 1.2\dot{q}_1 \\ 1.4\dot{q}_2 \\ 0.8\dot{q}_3 \end{bmatrix} \quad d = \begin{bmatrix} 3 \\ 5 \\ 4 \end{bmatrix} \quad (7.8.2)$$

The tracking errors when the model used to train the networks is not completely correct are off course a bit larger then when it was used a correct model to train the networks. However the tracking is still quite accurate and the results obtained with the different methods to implement the system dynamics are not very large. Only one figure of the joint trajectory is included since all the plots look the same. The different sum squared tracking errors will however be specified for both the whole period and for the last 2π simulation seconds only. This is the last whole sinus wave of the trajectory and the tracking error for this part alone is found to see how the networks performs when they have experienced some learning.

In figure 7.9 the desired trajectory and the actual trajectory for each joint can be seen for the situation where the system dynamics are implemented with mixed method pruned RBF networks.

7.8.1 Incorrect Model and Unpruned Networks

Whit friction and a constant disturbance present in the manipulator the known model is no longer entirely correct and implementing this in the controller would thus give a bit larger tracking error than for the case with no friction and disturbance present. The sum squared tracking error for the whole simulation was found to be 0.052887 and for the last 2π simulation time 0.0072445.

For the situation where the system dynamics are implemented with unpruned radial basis function neural networks as in table 7.1 the final tracking error was 0.052977 for the whole period and 0.0072385 for the last whole sinus wave of the trajectory.

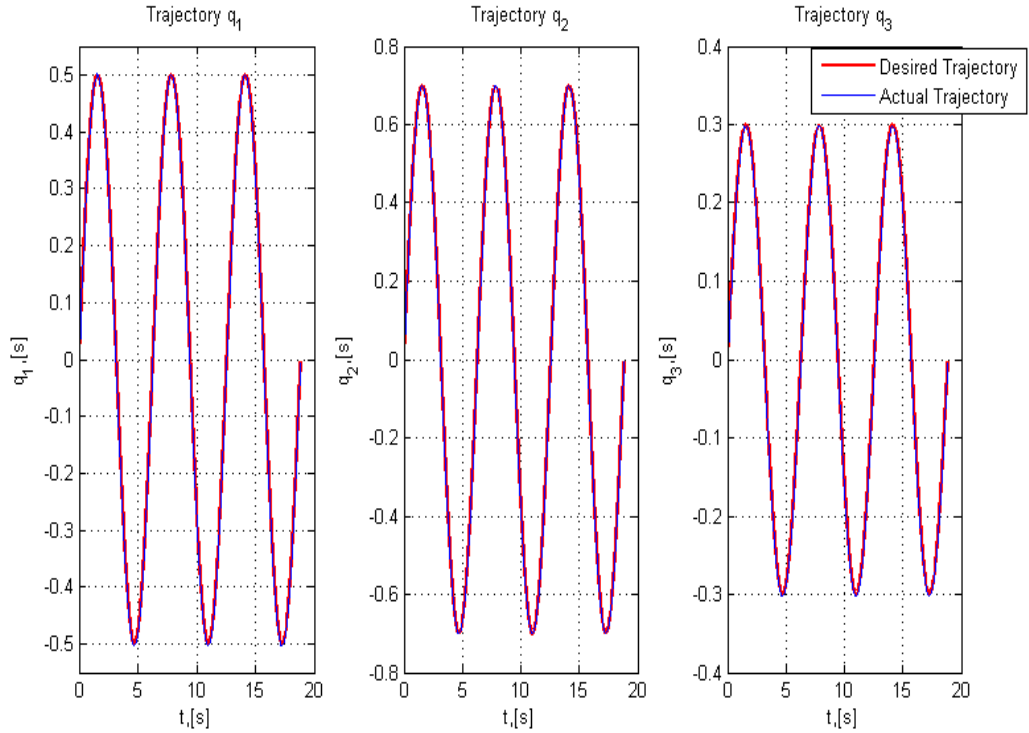


Figure 7.9: Friction and Disturbance - Trajectory all 3 Joints Using Mixed Methods Pruning

It can then be seen that the total tracking error for using the incorrect model is slightly better than using the unpruned networks while the tracking error for only the last 2π simulation time is a little bit better for the RBFNNs without pruning.

The unpruned networks have a total of 3672 hidden units in the 16 different networks.

7.8.2 Online Pruning

In the situation where the RBF networks now are pruned again the root mean squared approximation error has to be smaller than 0.01 for all the networks except the network \hat{g}_2 which must have an approximation error less than 0.05 before the RBFNNs can be pruned. This approximation error is found over a moving window of the 400 last simulation iterations.

The manipulator must also have completed one half of the sinus wave trajectory, π simulation time, before the weight pruning begins. For the pruning based

on neuron output ratio the manipulator has finished one whole period of the sinus wave, that is 2π of the simulation time. This was also the case in the last section without friction and disturbance.

Weight Magnitude Pruned Networks

All the RBF networks are pruned with 5% weight magnitude based pruning where the threshold again is growing. It can at most grow 5 times and the highest possible threshold in percent of the weights norm is thus 5×5 .

After the simulation has completed 4π simulation time it is no longer possible to prune the networks. This is done as in the previous section in order to make sure the networks are not over pruned due to the fact that the weights not converge.

The final networks after 6π simulation time is the same as they were when there were no friction or disturbance in the manipulator and the final sizes can be seen in table 7.3. In total there were 74 hidden neurons.

For the whole simulation the SSE tracking error was obtained as 0.052898 while it for the last 2π time of the simulation was found to be 0.0072129. These two tracking errors are somewhat smaller than for the case where it was used unpruned RBF networks instead.

Node Output Pruned Networks

When the dynamics of the manipulator is implemented with RBF networks pruned with neuron output based pruning it is used a threshold of 30% of the activation function ratio sum. This is the same as when the manipulator was without friction and disturbance but all the final networks here are not equal to the networks obtained in that case. The final networks \hat{c}_{12} and \hat{c}_{13} are both one node smaller in this section and the total sum of hidden units is now 124 instead of 126 as it was in the situation with no friction or disturbance.

All the final networks can be seen in table 7.7 below.

The reason for some of the final networks here being a bit different is that the inputs to the networks no longer are the same due to the friction and disturbance present. This again gives activation function ratio sums unequal to the ones obtained with no friction and disturbance. Since the inputs not are much changed only two networks have experienced a different pruning.

In this case a tracking error of 0.052975 for the whole simulation and 0.0072379 for the 2π simulation seconds at the end of the simulation are obtained. This is higher errors than for weight based pruning but smaller than for not pruning the networks. The tracking error for the last part of the trajectory is also smaller than using the incorrect model.

Friction and Disturbance Node Output Pruning	
<i>Network</i>	<i>Network Size</i>
\hat{m}_{11}	5
\hat{m}_{12}	5
\hat{m}_{13}	5
\hat{m}_{22}	2
\hat{m}_{23}	2
\hat{m}_{33}	2
\hat{c}_{11}	17
\hat{c}_{12}	21
\hat{c}_{13}	21
\hat{c}_{21}	11
\hat{c}_{22}	4
\hat{c}_{23}	7
\hat{c}_{31}	6
\hat{c}_{32}	6
\hat{g}_1	5
\hat{g}_2	5
Sum	124

Table 7.7: Friction and Disturbance - Node Output Pruning

Mixed Methods Pruned Networks

The RBF networks are also pruned with the same mixed methods as in the last section. It is used 7% and 5% weight based pruning for the \hat{M} networks and the \hat{G} networks respectively while it is used 30% node output pruning for the \hat{C} networks. The latter is the same as was used when all the networks were pruned with node output pruning and the same result regarding the different sizes of the \hat{C} networks was obtained here. Both the two networks for \hat{c}_{12} and \hat{c}_{13} had one unit less than they had after pruning in the control of a manipulator without friction and disturbance. For the rest of the networks they all are the same as

after being pruned with the mixed method in the last section. In table 7.8 all the final networks can be seen.

The figure 7.9 of the trajectories included earlier in this section are obtained with networks.

After the whole simulation of 6π the total tracking error is 0.052563 while for the last 2π simulation seconds it is 0.0071018. Compared to the incorrect model, unpruned networks and also the pruned networks from both weight based pruning and neuron output pruning alone these tracking errors are the smallest.

Friction and Disturbance Mixed Method Pruning	
<i>Network</i>	<i>Network Size</i>
\hat{m}_{11}	4
\hat{m}_{12}	4
\hat{m}_{13}	2
\hat{m}_{22}	3
\hat{m}_{23}	3
\hat{m}_{33}	2
\hat{c}_{11}	17
\hat{c}_{12}	21
\hat{c}_{13}	21
\hat{c}_{21}	11
\hat{c}_{22}	4
\hat{c}_{23}	7
\hat{c}_{31}	6
\hat{c}_{32}	6
\hat{g}_1	10
\hat{g}_2	6
Sum	127

Table 7.8: Friction and Disturbance - Mixed Method Pruning

7.8.3 Resource Allocation Networks EKF

The growing resource allocation networks extended kalman filter (RANKEF) is now implemented with the same thresholds as for the case of no friction or disturbance.

In table 7.9 the final networks after the 18.85 simulation seconds can be seen. The total sum of units is here 78 while it for the situation with no friction or disturbance was 129 hidden nodes in total. This is a huge difference in the networks for a small change in the system dynamics. One reason for this difference is that having fewer hidden units would give better generalization which is needed here.

This simulation gave the worst tracking error for the whole period of all the cases where friction and disturbance were included. It was found to be 0.053116 while the tracking error for the last sinus wave of the trajectory was 0.007243. The latter is worse than all the other cases except for the incorrect model.

Friction and Disturbance RANEKFs	
<i>Network</i>	<i>Network Size</i>
\hat{m}_{11}	2
\hat{m}_{12}	3
\hat{m}_{13}	2
\hat{m}_{22}	1
\hat{m}_{23}	1
\hat{m}_{33}	1
\hat{c}_{11}	9
\hat{c}_{12}	9
\hat{c}_{13}	9
\hat{c}_{21}	8
\hat{c}_{22}	4
\hat{c}_{23}	3
\hat{c}_{31}	3
\hat{c}_{32}	3
\hat{g}_1	18
\hat{g}_2	2
Sum	78

Table 7.9: Friction and Disturbance - RANEKFs

7.9 Ill-Posed Learning and Weight Convergence

In the section about weight based pruning it was mentioned that the norm of the weights not converged and thus the pruning did not end even if the threshold

stopped growing. Since the norm of the weights not converge it follows that the individual weights for the networks not converge.

For neural network training it is crucial that the weights converge to a value for the training to be completed. Here the situation however is found to be an *ill-posed* learning problem. For a brief explanation of what an ill-posed problem is see chapter 2.

The learning here is ill-posed due to the training data set contains to little information for the networks to be able to experience enough varied training. Since increasing the simulation length only repeated the same training data the information was not sufficient to complete the training for the networks. Hence the weights did not converge properly. Increasing the simulation length gave however that the networks approximation errors decreased and thus the learning is not very ill-posed.

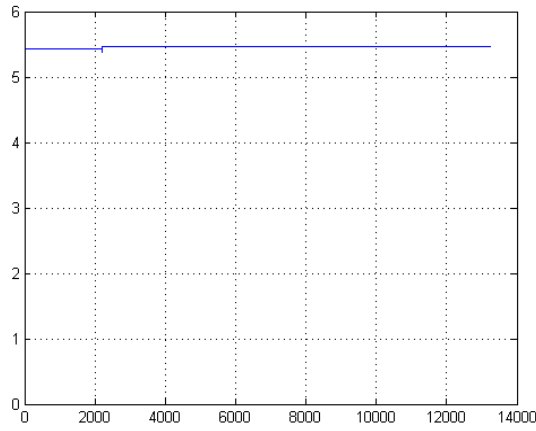
There was however one exception and that was the network \hat{m}_{33} . This network has the input 1 since the element m_{33} only is a constant. The training for this network do not need much information since it is simple and the weights here converge extremely fast as can be seen in figure 7.10a. In this figure it can be seen that there is a small change in the norm after some time. This is when the network was pruned with 30% node output pruning and the remaining weights were changed. Even if the network was reduced from 9 hidden units to only 2 hidden nodes the norm stayed approximately constant due to the remaining weights having their magnitude increased. For this figure the simulation time was 6π .

Along the x-axis is the iteration number shown for all the plots in this section while the value of the norm at the different iterations is shown along the y-axis.

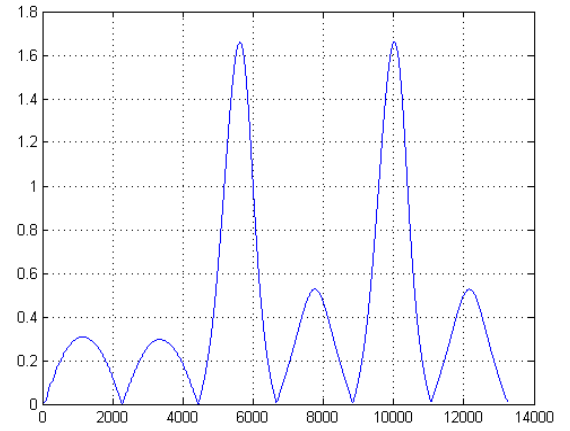
In figure 7.10b the norm for the network \hat{m}_{13} when the network is pruned with weight based pruning is also shown. Simulation time was 6π . As visible from the figure the norm oscillates very much. The amplitude of the oscillations were however not the same for the weight magnitude pruned and the node output pruned networks.

A figure of the norms for the two \hat{G} networks when the networks are pruned with 30% neuron output based pruning and the simulation had the length 20π is also included. These are shown in figure 7.11 below. The norm of the weights oscillates in different ways for these two networks. For network \hat{g}_2 the amplitude is very large for some of the waves while for the network \hat{g}_3 the amplitude is much smaller.

For the \hat{C} networks it is found that the norms not oscillates as nicely like for \hat{M}

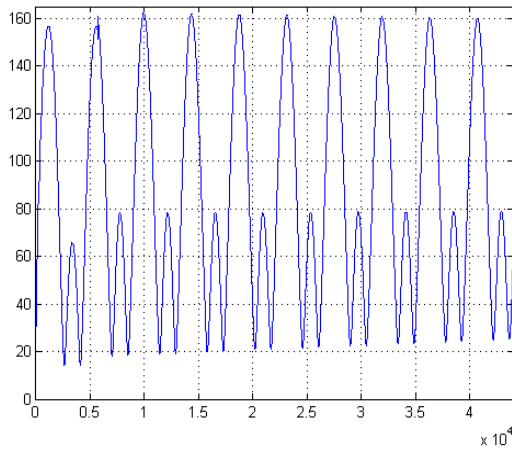


(a) \hat{m}_{33} Node Output Pruned

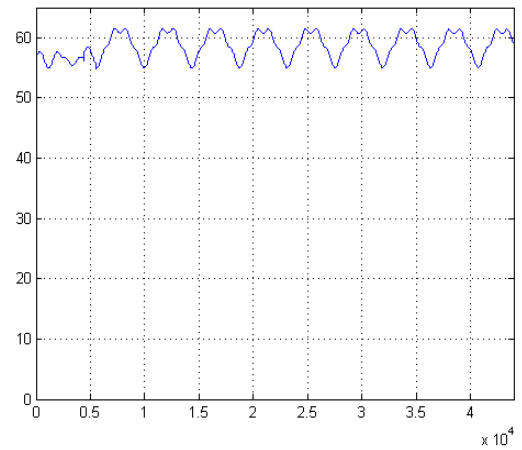


(b) \hat{m}_{13} Weight Based Pruned

Figure 7.10: Norm of Weights some \hat{M} Networks



(a) \hat{g}_2 Node Output Pruned



(b) \hat{g}_3 Node Output Pruned

Figure 7.11: Norm of Weights \hat{G} Networks

and \hat{G} . This can be seen from the plots in figure 7.12. Here the norm of network \hat{c}_{21} pruned with either node output pruning 7.12a or weight magnitude pruning 7.12b are shown. Both norms for the pruned and the not pruned \hat{C} networks do not start to oscillate before the simulation has done more than 20π simulation second. The oscillations are not even and not smooth.

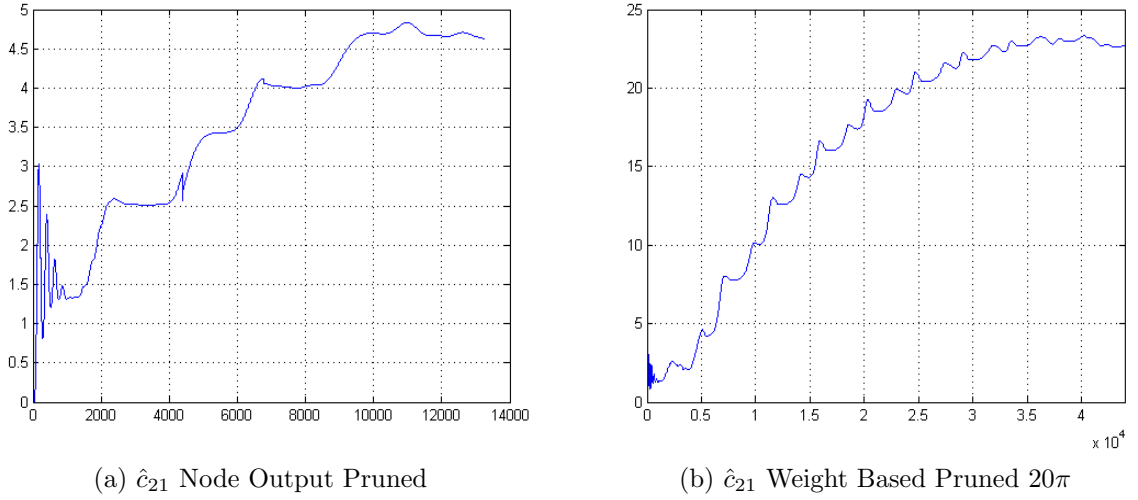


Figure 7.12: Norm of Weights Different \hat{c}_{23} Networks

In figure 7.12a the network \hat{c}_{21} is pruned with 30% node output pruning in a simulation that lasted 6π simulation seconds. Here it can be seen that the norm grows up to almost 5 before it starts to decrease again. In the figure 7.12b the network is pruned with weight based pruning with 20π simulation time. This plot shows that the norm behaves very differently from when the network was pruned with node output pruning. The norm do not start to decrease as fast after being pruned with weight pruning and it actually grows until it almost reaches 25.

The \hat{C} networks have more inputs and some of the variables, q and \dot{q} , are added together into one input to the network in order to reduce the number of dimensions. Thus it could be that the learning for these networks are more ill-posed than for the \hat{M} and \hat{G} networks depending on fewer variables. Adding of the inputs is not found to create any larger ill-posing than having them separately and the network \hat{c}_{21} in the figure 7.12 has 3 different inputs that not are added together. In the beginning of the simulation it looks like if the norm starts to converge before it grows a lot instead.

From the above mentioned facts it is obvious that having a pruning threshold depending on this norm would give uncertainties regarding how many times the networks are pruned and how many neurons that will be removed in total. Since the norm always starts to decrease again at some point the pruning will also stop by itself when the threshold in percent stops growing. This may however take many simulation seconds as it would in the case with weight pruning for the network \hat{c}_{21} . Hence it is very likely that some networks will be pruned to much. In the

previous section it was specified a stopping time in the simulation for the pruning to avoid removing too many units.

In chapter 3 it was briefly described *regularization* in the literature review since it often is mentioned together with pruning. However originally it was developed for ill-posed problems. In regularization a regularizing term to the cost function that is minimized during training of a network is included. The cost function then becomes $\tilde{E} = E + \lambda\Omega$, where E is the standard error term and $\lambda\Omega$ is the regularizing term. The regularization factor, λ , is an indicator of the sufficiency of the information in the training data set. A problem that is highly ill-posed will have a large regularization factor while if the problem is completely well-posed this term goes to zero and the training is as normal. How the term *Omega* is defined is varied and one example can be seen in the literature review.

7.10 Discussion

7.10.1 Results

In table 7.10 it is summarized the results from using the different methods to implement the system dynamics in the controller when the simulation time was 6π . For each of the method the final number of hidden units for all the networks can be seen together with the tracking sum squared error for the whole simulation, the approximation error for the last 400 iterations of the simulation and also the time it actually took to complete the 18.85 simulation seconds.

From this table it can be seen that all the methods gave almost the same tracking error and that using the correct model for the manipulator obtained the best tracking. The unpruned networks gave better tracking and smaller approximation error than using pruned networks. It took however more time to finish the simulation with unpruned networks.

The smallest networks were obtained when using weight magnitude pruning for the RBFNNs while the node output pruning gave several more hidden neurons. Weight magnitude pruning resulted however in larger tracking error and larger approximation error than the node output pruned networks did. Mixing the two methods and using weight pruning for \hat{M} and \hat{G} while using node output for \hat{C} gave a few more hidden neurons than node output pruning and smaller tracking and approximation errors. The quickest method when using RBF networks was the weight based pruning, then the mixed method and neuron output used the same amount of time and using the networks without pruning took most time.

In this chapter the growing RBF networks RANEKFs has also been implemented. As can be seen from table 7.10 gave RANEKFs a much smaller approximation error for the last part of the simulation. The estimations were probably not so accurate in the beginning while the RANEKFs had few neurons and the largest tracking error for the whole simulation came from these networks. Growing the networks must also be a time consuming process as is visible when looking at the long time this method used.

Trajectory Tracking and Network Approximation Errors				
6π Simulation Time				
<i>Method</i>	<i>Sum of Hidden Units</i>	<i>Tracking Error Whole Sim. [SSE]</i>	<i>Sum Approx. Errors [RMSE]</i>	<i>Time [min]</i>
Correct Model	-	0.031897	-	0.25
Unpruned Networks	3672	0.032	0.26303	4.49
Weight Based Pruning	74	0.03201	0.29037	4.10
Neuron Output Pruning	126	0.032002	0.28021	4.20
Mixed Methods Pruning	129	0.032001	0.26314	4.20
RANEKF	96	0.032094	0.024542	7.20

Table 7.10: Trajectory Tracking and Approximation Errors 6π Simulation Time

Table 7.11 shows the same as the previous table except that the simulation time here was 20π and thus much longer than the last. The pruned networks all had the same amount of hidden nodes at the end of simulation and a smaller approximation error. Also the unpruned networks had a better approximation error that still was smaller than the one from any of the pruned networks. This shows that the networks give better estimations when they have experienced some more training.

The RANEKFs however do not have the same number of hidden neurons as after the shorter simulation. Here the networks have grown a lot more and also the approximation error at the end of this simulation is worse than after the shorter simulation. Still this approximation error is far better than for any of the other RBF networks. RANEKFs however are developing in the wrong direction and the networks start to overfit the training data. Again RANEKF is the method with the longest time to finish the simulation. It seems however that the most time consuming part for the RANEKFs is the beginning of the simulation since the gap between the pruning methods and the RANEKFs not is much bigger after 20π simulation time than after 6π simulation time.

Trajectory Tracking and Network Approximation Errors 20π Simulation Time				
<i>Method</i>	<i>Sum of Hidden Units</i>	<i>Tracking Error Whole Sim.[SSE]</i>	<i>Sum Approx. Errors [RMSE]</i>	<i>time [min]</i>
Correct Model	-	0.031897	-	1.44
Unpruned Networks	3672	0.032015	0.24867	25.30
Weight Based Pruning	74	0.032053	0.28429	23.30
Neuron Output Pruning	126	0.0320025	0.2657	23.25
Mixed Methods Pruning	129	0.032018	0.25074	23.40
RANEKF	147	0.032095	0.058592	27.05

Table 7.11: Trajectory Tracking and Approximation Errors 20 π Simulation Time

All of the methods of implementing the learning inverse kinematic controller were also used when there was added some friction and constant disturbance to the manipulator. The obtained results from this can be seen in table 7.12. In this table the total number of hidden nodes for all the networks, the tracking error for the whole period and also the tracking error for only the last 2 π simulation time are shown.

Here it can be seen that the tracking errors all are worse than for the case without friction and disturbance. This is due to the model of the manipulator that is used in the controller and also in the training of the networks now contains modelling errors.

The sizes for the networks are the same after weight magnitude pruning as they were when the manipulator was without friction and disturbance while the node output pruning and mixed methods both have two hidden units less. For the mixed methods these two units come from the networks that are pruned with node output pruning. This is due to the inputs no longer being the same and thus the node output ratio sums are not the same. RANEKFs do also have less hidden neurons and for these networks the differences in hidden layers are larger.

For the whole simulation the best tracking is obtained with the RBF networks that are pruned with the mixed method. This method gives better tracking than using the know incorrect model directly in the controller. When looking at the tracking error for the last sinus wave of the manipulator trajectory it can be seen that it again is the mixed method that has the smallest error. Since the networks have been trained a bit for this last part of the simulation they all give better tracking than using the incorrect model. It can also be seen that the pruned networks now gives better tracking than the RBF networks not pruned. This is

probably due to the pruned networks having a better generalization ability than the larger networks.

Weight based pruning gave here a better result than node output pruning for both the whole period and the last 2π simulation seconds only. Neuron output pruning do as known from previous chapters not remove neurons that are close to the inputs. This gives that node output pruned networks are better in approximation when the training data are correct as was shown in the last table. Weight magnitude pruning can also remove units that are close to the inputs due to there being more neurons than necessary in an area. Thus the weight magnitude pruned networks were smaller in size and from the simulations with friction and disturbance they were also found to be better in generalization.

The worst tracking error out of the different RBF networks came again from the RANEKFs. When looking at the whole simulation these networks did worse than all the other methods. At the end of the simulation when the networks had grown they did however give better tracking than the model with modelling errors. This may suggest that the generalization ability of the RANEKFs is less than for pruned networks.

Disturbance and Friction			
Trajectory Tracking and Network Approximation Errors			
<i>Method</i>	<i>Sum of Hidden Units</i>	<i>Tracking Error Whole Sim. [SSE]</i>	<i>Tracking Error Last 2π Time[SSE]</i>
Incorrect Model	-	0.052887	0.0072445
Unpruned Networks	3672	0.052977	0.0072385
Weight Based Pruning	74	0.052898	0.0072129
Neuron Output Pruning	124	0.052975	0.0072379
Mixed Methods Pruning	127	0.052563	0.0071018
RANEKF	78	0.053116	0.007243

Table 7.12: Disturbance and Friction - Trajectory Tracking and Approximation Errors

From this it is clear that using RBF networks make the controller more robust towards modelling errors. Pruning the networks enhanced their generalization ability and made the inverse kinematic controller better for dealing with uncertainties.

The growing networks are much better in giving accurate approximations when the model used to train them is correct. When the simulation time is increased

they do however keep on growing and increases their estimation errors. A problem with their accuracy is also that they seem to be poorer when it comes to generalization. This suggest that RANEKFs very quickly overfit the training data. These growing networks are also a lot more time consuming.

7.10.2 Simulation Time

In the results the time the simulation actually take to complete is given. This is however not measured very accurately and there are probably some minor errors. It has however been used the same computer for all simulations and the amount of other programs running on the computer at the same time have been approximately equal. Thus he time measurement is possible to use for comparing the different methods. The dimension would give a hint of what method that uses most computational time.

Radial basis function neural networks with their structure are very well suited for use in parallel processing. All of the hidden units can be processed simultaneously which would give much faster computations. Here many small networks is used which also would make it more possible to utilize parallel processing for making it more usable to real-time control.

In a master's thesis [5] to use a Graphic Processing Unit (GPU) for calculations with neural networks is proposed. Here it is found that a GPU may be better suited for the small parallel tasks of a neural network than the normal Central Processing Unit, CPU.

7.10.3 Ill-Posed Problem

In this chapter the learning problem is found to be an ill-posed one. The training data do not contain sufficiently information on the desired solution for the training to be complete and weights to converge to an optimal value. Even if the weights do not converge they do not grow for ever. All the norms of the weights oscillated in some way.

Implementing the weight magnitude pruning that removes units based on the norm of the weights may give some problems when this norm varies a lot. Stopping the pruning threshold in percent from growing is not necessarily the same as stopping the actual threshold from growing when the norm increases. Thus when the pruning should end can be specified as was done in this chapter to prevent the networks from being pruned too much.

It is also found that networks dependent on many variables are more ill-posed than the networks with fewer variables as inputs. For the situation in this chapter the networks \hat{C} became more ill-posed due to the many joint positions and joint velocities the elements in the centrifugal matrix were depending on. Pruning of these networks with neuron output pruning gave thus the best result.

For the other networks, \hat{M} and \hat{G} , that estimated the inertia matrix and the gravitational matrix the weight based pruning did however give a good result.

When the number of joints increases the elements in the dynamic model matrices become dependent on more variables. If it should be included the last 3 joints of the ABB IRB140 in the controller this would give networks with very many dependencies, especially for the centrifugal matrix C . This again would make the learning even more ill-posed. Thus it should be tried with regularization before including more of the joints in the controller.

Chapter 8

Concluding Remarks and Future Work

8.1 Conclusion

In this report two different pruning methods for radial basis function neural networks, Weight Magnitude Pruning and Neuron Output Pruning, suited for use in an online learning controller for robot manipulators have been proposed. Implementation of pruning in the learning controller made it easier to create suitable RBF networks. The model-based learning controller is also more robust towards system uncertainties than the pure model-based controller. By pruning the RBF networks the generalization abilities were increased and the obtained networks performed better in the case of modelling errors. Pruning also reduced the computational cost for the networks.

Some essential background information on radial basis function neural networks and an overview of some of the different pruning schemes existing today have in this report been given. Theory on the proposed pruning methods and a general algorithm are also presented. Simulations with the two pruning methods for first learning of a cross function and then for learning the dynamics of a 2 dof manipulator have been conducted. Finally the pruning methods were implemented in a learning controller for the first 3 joints of the ABB IRB140 manipulator.

For both pruning methods a threshold specified as a percent and thus less problem dependent to find was developed. This made it possible to only give one threshold for pruning off all the different networks.

From the simulations the weight based pruning and the node output pruning was found to not always remove the same nodes. Neuron output pruning often

gave networks with a good approximation ability since all the neurons close to the inputs were kept. Weight based pruning had the ability to remove neurons if there were too many units in an area even if this area was close to the inputs. This resulted in smaller networks better to generalize.

Which method that was best suited depended on the situation. Thus a better result in total by looking at the different networks separately could be obtained.

Networks with more hidden units often had more accurate estimations. The question then is how important to reduce the network sizes as much as possible is or if larger networks can be accepted. This again would be dependent on the situation and what the networks are implemented on.

8.2 Further Work

NTNU has a manipulator of the type ABB IRB140 and the next step would be to implement the learning inverse kinematic controller on this robot.

When discussing the ill-posed learning situation for the ABB IRB140 a known solution to ill-posed problems was mentioned to be regularization. Thus including regularization in the training to see if the learning then becomes well-posed should be done.

Bibliography

- [1] Chae H. An, Christopher G. Atkeson, and John M. Hollerbach. Model-based control of a direct drive arm, part 2: Control. *1988 IEEE International Conference on Robotics and Automation*, 3.
- [2] Somwang Arisariyawong and Siam Charoenseang. Dynamic self-organized learning for optimizing the complexity growth of radial basis function neural networks. *IEEE International Conference on Industrial Technology, 2002. IEEE ICIT '02.*, 1.
- [3] Mohammed Attik, Laurent Bougrain, and Frédéric Alexandre. Optimal brain surgeon variants for feature selections. *Proceedings 2004 IEEE International Joint Conference on Neural Networks 1993*, 2.
- [4] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal brain damage. *Proceedings of the Neural Information Processing Systems*.
- [5] Christopher Edward Davis. Graphics processing unit computation of neural networks. Master's thesis, University of Mexico, 2001.
- [6] Sigurd A. Fjerdings, Magnus Bjerken, Aksel A. Transeth, Erik Kyrkjebø, and Anders Røyroy. A learning camera platform for remote operations with industrial manipulators.
- [7] S.S. Ge, T.H. Lee, and C.J. Harris. *Adaptive Neural Network Control of Robotic Manipulators*. World Scientific Publishing Co, Singapore, 1998.
- [8] Masafumi Hagiwara. Removal of hidden units and weights for back propagation network. *Proceedings of 1993 International Joint Conference on Neural Networks, 1993. IJCNN '93-Nagoya.*, 1.
- [9] Babak Hassibi, David G. Stork, and Gregory J. Wolff. Optimal brain surgeon and general network pruning. *IEEE International Conference on Neural Networks 1993*.

- [10] Simon Haykin. *Neural Networks – A Comprehensive Foundation*. Pearson - Prentice Hall, ninth reprint 2005 edition, 1999.
- [11] Ralph E. Hoffman and Thomas H. McGlashan. Neural network models of schizophrenia. *The Neuroscientist*, 2001.
- [12] Han Honggui and Qiao Junfei. A novel pruning algorithm for self-organized neural network. *IJCNN 2009 International Joint Conference on Neural Network*.
- [13] William W. Hsieh and Benyang Tang. Applying neural network models to prediction and data analysis in meteorology and oceanography. *Bulletin of the American Meteorological Society*, 79(9):1855–1870, May 1998.
- [14] Thuan Q. Huynh and Rudy Setiono. Effective nn pruning using cross-validation. *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005. IJCNN '05.*, 2.
- [15] Visakan Kadiramanathan and Mahesan Niranjan. A function estimation approach to sequential learning with neural networks. *Neural Computation*, 5.
- [16] Marcin Kaminski and Teresa Orłowska-Kowalska. Comparison of bayesian regularization and optimal brain damage methods in optimization of neural estimators for two-mass drive system. *2010 IEEE International Symposium on Industrial Electronics (ISIE)*.
- [17] Hassan K. Khalil. *Nonlinear systems*. Prentice-Hall, third, international edition, 2000.
- [18] M. Vidyasagar Mark W. Spong, Seth Hutchinson. *Robot Modeling and Control*. John Wiley and Sons, INC.
- [19] Jiang Meng. Penalty obs scheme for feedforward neural network. *ICTAI 05. 17th IEEE International Conference on Tools with Artificial Intelligence, 2005*.
- [20] Kieron Messer and Josef Kittler. Fast unit selection algorithm for neural network design. *Proceedings 15th International Conference on Pattern Recognition, 2000*, 2.
- [21] Michael C. Mozer and Paul Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. *Advances in Neural Information Processing*.

- [22] Jie Ni, Qing Song, and M.J. Grimble. Robust pruning of rbf network for neural tracking control systems. *Proceedings of the 45th IEEE Conference on Decision and Control*.
- [23] Mark J. L. Orr. Introduction to radial basis function networks. pages 1–67, April 1996.
- [24] J. Park and I. W. Sandberg. Universal approximation using radial-basis-function networks. *Neural Comput.*, 3(2):246–257, 1991.
- [25] Jung-Wok Park, R.G. Harley, and G.K. Venayagamoorthy. Comparison of mlp and rbf neural networks using deviation signals for on-line identification of a synchronous generator. *IEEE Power Engineering Society Winter Meeting, 2002*, 1.
- [26] John Platt. A resource-allocating network for function interpolation. *Neural Computation*, 3.
- [27] P.V.S. Ponnappalli, K.C. Ho, and M. Thomson. A formal selection and pruning algorithm for feedforward artificial neural network optimization. *IEEE Transactions on Neural Networks*, 10(4):964–968, July 1999.
- [28] Devin Sabo and Xiao-Hua Yu. A new pruning algorithm for neural network dimension analysis. *IEEE International Joint Conference on Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence)*.
- [29] Achim Stahlberger and Martin Riedmiller. Fast network pruning and feature extraction using the unit-obs and algorithm. *Advances in Neural Information Processing Systems*.
- [30] Zhaozhao Zhang and Junfei Qiao. A novel pruning algorithm for feedforward neural network based on neural complexity. *2010 International Conference on Intelligent Control and Information Processing (ICICIP)*.
- [31] Shouling Zhao, Quan Liu, and Binbin Zhang. A fast obs pruning algorithm based on pseudo-entropy of weights. *2010 2nd International Conference on Advanced Computer Control (ICACC)*, 3.

Appendix A

Stability Proves

A.1 Stability of Inverse Kinematic Controller

Here it is proven that the inverse kinematic controller used in the simulation is Globally Asymptotically Stable (GAS) when the matrices in the dynamics equations of the manipulator are used. The equilibrium points considered are

The controller is given as in equation (6.3.1) and restated here

$$\tau = M(q)(\ddot{q}_d + K_d(\dot{q}_d - \dot{q}) + K_p(q_d - q)) + C(q, \dot{q})\dot{q}_d + G(q) \quad (\text{A.1.1})$$

Defining the error variable

$$e \triangleq q_d - q \quad (\text{A.1.2})$$

where q_d is the desired trajectory and q is the actual joint position.

It is started with the following Lyapunov Function Candidate

$$V(e, \dot{e}) = \frac{1}{2}\dot{e}^T M(q)\dot{e} + \frac{1}{2}e^T K e \quad (\text{A.1.3})$$

which has $V(0, 0) = 0$, is positive definite and radially unbounded when it is assumed that $M(q) = M(q)^T$ and K both are positive definite.

The derivative of $V(e, \dot{e})$ is given as

$$\dot{V} = \dot{e}^T M \ddot{e} + \frac{1}{2}\dot{e}^T \dot{M} \dot{e} + e^T K \dot{e} \quad (\text{A.1.4})$$

Using that $\ddot{e} = \ddot{q}_d - \ddot{q}$ and inserting this gives

$$\dot{V} = \dot{e}^T M \ddot{q}_d - \dot{e}^T M \ddot{q} + \frac{1}{2}\dot{e}^T \dot{M} \dot{e} + e^T K \dot{e} \quad (\text{A.1.5})$$

$$= \dot{e}^T M \ddot{q}_d - \dot{e}^T (\tau - C\dot{q} - G) + \frac{1}{2}\dot{e}^T \dot{M} \dot{e} + e^T K \dot{e} \quad (\text{A.1.6})$$

where it in the last equation has been used equation (2.6.1). Then inserting that $\dot{q} = \dot{q}_d - \dot{e}$ gives

$$\dot{V} = \dot{e}^T M \ddot{q}_d - \dot{e}^T (\tau - C \dot{q}_d + C \dot{e} - G) + \frac{1}{2} \dot{e}^T \dot{M} \dot{e} + e^T K \dot{e} \quad (\text{A.1.7})$$

$$= \dot{e}^T (M \ddot{q}_d - \tau + C \dot{q}_d + G + K e) + \frac{1}{2} \dot{e}^T (\dot{M} - 2C) \dot{e} \quad (\text{A.1.8})$$

$$(\text{A.1.9})$$

Using the skew symmetric property of $\dot{M} - 2C$ and inserting for τ with equation (A.2.1) it is obtained

$$\dot{V} = \dot{e}^T (M \ddot{q}_d - (M(\ddot{q}_d + K_d(\dot{q}_d - \dot{q}) + K_p(q_d - q)) + C \dot{q}_d + G) + C \dot{q}_d + G + K e) \quad (\text{A.1.10})$$

$$= \dot{e}^T (-(M(K_d \dot{e} + K_p e) + K e)) \quad (\text{A.1.11})$$

$$= -\dot{e}^T M K_d \dot{e} - \dot{e}^T (M K_p - K) e \quad (\text{A.1.12})$$

$$(\text{A.1.13})$$

If K_d is chosen to be positive definite and K_p is chosen such that $M K_p - K = 0$ then

$$\dot{V} = -\dot{e}^T M K_d \dot{e} \leq 0 \quad (\text{A.1.14})$$

is negative semi-definite and it can be concluded that the controller in equation (6.3.1) will make the equilibrium point $(e, \dot{e}) = 0$ stable according to Theorem 4.1 in [17].

In order to show asymptotic stability LaSalle's theorem is applied. It is then started by defining the set

$$E = \{(e, \dot{e}) | \dot{V} = 0\} = \{(e, \dot{e}) | \dot{e} = 0\} \quad (\text{A.1.15})$$

An error equation for the system can be found by starting with the dynamic equation from (2.6.1) and then inserting with the controller from (A.2.1)

$$M(q) \ddot{q} + C(q, \dot{q}) \dot{q} + G(q) = \tau \quad (\text{A.1.16})$$

$$M \ddot{q} + C \dot{q} + G = M(\ddot{q}_d + K_d(\dot{e}) + K_p e) + C \dot{q}_d + G \quad (\text{A.1.17})$$

From (A.1.2) it is now used $q = q_d - e$ and the following is obtained

$$-M \ddot{e} + M \ddot{q}_d - C \dot{e} + C \dot{q}_d + G = M(\ddot{q}_d + K_d(\dot{e}) + K_p e) + C \dot{q}_d + G \quad (\text{A.1.18})$$

$$-M \ddot{e} - C \dot{e} = M(K_d(\dot{e}) + K_p e) \quad (\text{A.1.19})$$

where the last step is from cancelling terms.

When (e, \dot{e}) is within the set E it follows from (A.1.19) that the solution will converge to the invariant set $M = (0, 0)$. Thus it can be concluded by LaSalle's theorem that the inverse kinematic controller will make the equilibrium point $(e, \dot{e}) = (0, 0)$ globally asymptotically stable (GAS). For a trajectory that starts close to the desired trajectory it then follows that the actual trajectory will converge to the desired one and stay there.

A.2 Stability of Learning Inverse Kinematic Controller

The stability proof here is somewhat similar to the ones in to the book by Ge, Lee and Harris [7].

A.2.1 RBF Networks for Learning System Dynamics

Now the system dynamics matrices are approximated by Radial Basis Function Neural Networks.

The controller is then given as

$$\tau = \hat{M}(q)(\ddot{q}_d + K_d(\dot{q}_d - \dot{q}) + K_p(q_d - q)) + \hat{C}(q, \dot{q})\dot{q}_d + \hat{G}(q) \quad (\text{A.2.1})$$

where \hat{M} , \hat{C} and \hat{G} are estimations of M , C and G respectively. They are given as in chapter 2.5,

$$\hat{M} = W_M^T \bullet A_M(x) \quad (\text{A.2.2})$$

$$\hat{C} = W_C^T \bullet A_C(x) \quad (\text{A.2.3})$$

$$\hat{G} = W_G^T \bullet A_G(x) \quad (\text{A.2.4})$$

where W_M is the weight matrix and $A_M(x)$ is the activation function matrix to input x , both for the RBF networks belonging to \hat{M} . \hat{C} has the weight matrix W_C and the activation function matrix $A_C(x)$ for input x while \hat{G} has the weight matrix W_G and the activation function matrix $A_G(x)$ to the same input.

The matrices can be described with an optimal estimation and an estimation error as

$$M = M^* + E_M = W_M^{*T} \bullet A_M(x) + E_M \quad (\text{A.2.5})$$

$$C = C^* + E_C = W_C^{*T} \bullet A_C(x) + E_C \quad (\text{A.2.6})$$

$$G = G^* + E_G = W_G^{*T} \bullet A_G(x) + E_G \quad (\text{A.2.7})$$

where M^* , C^* and G^* are the optimal estimations and W_M^* , W_C^* and W_G^* are the optimal weights and E_M , E_C and E_G are the estimation errors. From the theory on RBF networks these estimation errors can be made arbitrary small and thus they can be bounded as

$$E_M \leq E_M^* \quad (\text{A.2.8})$$

$$E_C \leq E_C^* \quad (\text{A.2.9})$$

$$E_G \leq E_G^* \quad (\text{A.2.10})$$

where $E_M^* > 0$ is the optimal approximation error for \hat{M}^* , $E_C^* > 0$ is the optimal approximation error for \hat{C}^* and $E_G^* > 0$ is the optimal approximation error for \hat{G}^* . All the optimal approximation errors are very small in size.

Since the learning rule used is the gradient descent the weights will converged to a minima. [10]. Thus the weights will always be bounded since they start at zero and do never grow to infinity due to convergence guarantee.

The differences from the optimal estimation and the estimation given from the networks are defined as

$$\tilde{M} \triangleq M^* - \hat{M} \quad (\text{A.2.11})$$

$$\tilde{C} \triangleq C^* - \hat{C} \quad (\text{A.2.12})$$

$$\tilde{G} \triangleq G^* - \hat{G} \quad (\text{A.2.13})$$

which are bounded due to the fact that both the optimal estimation and the actual estimation will be bounded from the weights and activation functions being bounded. As the networks are trained this difference will become very small.

As mentioned the update law for the weights is the gradient descent rule which also is given in (2.3.3). It is restated here with the approximation error for the network to the given input renamed to e_A

$$\Delta \mathbf{w} = -e_A \mathbf{a}(x) \quad (\text{A.2.14})$$

where $\mathbf{a}(x)$ is the activation function output for the input x .

Writing the approximation error in matrix form gives

$$E_A = \begin{bmatrix} e_{A11} & \cdots & e_{A1n} \\ \vdots & \ddots & \vdots \\ e_{An1} & \cdots & e_{Ann} \end{bmatrix} \quad (\text{A.2.15})$$

where n is the number of joints in the manipulator. Now the the update law from (A.2.14) may be written in matrix form as

$$\Delta W = \begin{bmatrix} -e_{A11}a_{11}(x) & \cdots & -e_{A1n}a_{1n}(x) \\ \vdots & \ddots & \vdots \\ -e_{An1}a_{n1}(x) & \cdots & -e_{Ann}a_{nn}(x) \end{bmatrix} = -E_A \bullet A(x) \quad (\text{A.2.16})$$

This can be done for all the estimated dynamic matrices as

$$\Delta W_M = -E_{AM} \bullet A_M(x) \quad (\text{A.2.17})$$

$$\Delta W_C = -E_{AC} \bullet A_C(x) \quad (\text{A.2.18})$$

$$\Delta W_G = -E_{AG} \bullet A_G(x) \quad (\text{A.2.19})$$

A.2.2 Error Equation

Now the tracking error variable is defined in the same way as for the case without neural networks, that is

$$e \triangleq q_d - q \quad (\text{A.2.20})$$

where q_d is the desired trajectory and q is the actual joint position.

An error equation can then be found starting with the dynamic equation (2.6.1)

$$M\ddot{q} + C\dot{q} + G = \tau \quad (\text{A.2.21})$$

$$\Rightarrow -M\ddot{e} + M\ddot{q}_d - C\dot{e} + C\dot{q}_d + G = \tau \quad (\text{A.2.22})$$

where the tracking error from (A.2.20) has been used. By inserting with the optimal estimations from RBF networks it is obtained

$$-M\ddot{e} + M^*\ddot{q}_d + E_M\ddot{q}_d - C\dot{e} + C^*\dot{q}_d + E_C\dot{q}_d + G^* + E_G = \tau \quad (\text{A.2.23})$$

Now is a new approximation error defined as

$$E \triangleq E_M\ddot{q}_d + E_C\dot{q}_d + E_G \quad (\text{A.2.24})$$

Using this approximation error and inserting with the controller from equation (A.2.1) it is obtained

$$-M\ddot{e} + M^*\ddot{q}_d - C\dot{e} + C^*\dot{q}_d + G^* + E = \hat{M}((\ddot{q}_d + K_d\dot{e} + K_p e) + \hat{C}\dot{q}_d + \hat{G}) \quad (\text{A.2.25})$$

$$\Rightarrow (M^* - \hat{M})\ddot{q}_d + (C^* - \hat{C})\dot{q}_d + (G^* - \hat{G}) + E = M\ddot{e} + C\dot{e} + \hat{M}K_d\dot{e} + \hat{M}K_p e \quad (\text{A.2.26})$$

$$\Rightarrow \tilde{M}\ddot{q}_d + \tilde{C}\dot{q}_d + \tilde{G} + E = M\ddot{e} + C\dot{e} + \hat{M}K_d\dot{e} + \hat{M}K_p e \quad (\text{A.2.27})$$

A.2.3 Lyapunov Function Candidate

It is started with the following Lyapunov Function Candidate

$$V(e, \dot{e}) = \frac{1}{2} \dot{e}^T M(q) \dot{e} + \frac{1}{2} e^T \tilde{W}_M^T \tilde{W}_M e \quad (\text{A.2.28})$$

which has $V(0, 0) = 0$, is positive definite and radially unbounded when it is assumed that $M(q) = M(q)^T$ is positive definite.

The derivative \dot{V} is

$$\dot{V} = \dot{e}^T M \ddot{e} + \frac{1}{2} \dot{e}^T \dot{M} \dot{e} + e^T \tilde{W}_M^T \tilde{W}_M \dot{e} + e^T \tilde{W}_M^T \dot{\tilde{W}}_M e \quad (\text{A.2.29})$$

Inserting for $M\ddot{e}$ with (A.2.27) gives

$$\dot{V} = \dot{e}^T (-C\dot{e} - \hat{M}K_d \dot{e} - \hat{M}K_p e + \tilde{M}\ddot{q}_d + \tilde{C}\dot{q}_d + \tilde{G} + E) + \frac{1}{2} \dot{e}^T \dot{M} \dot{e} + e^T \tilde{W}_M^T \tilde{W}_M \dot{e} + e^T \tilde{W}_M^T \dot{\tilde{W}}_M e \quad (\text{A.2.30})$$

$$\begin{aligned} &= \frac{1}{2} \dot{e}^T (\dot{M} - 2C) \dot{e} - \dot{e}^T \hat{M}K_d \dot{e} - \dot{e}^T \hat{M}K_p e + \dot{e}^T (\tilde{M}\ddot{q}_d + \tilde{C}\dot{q}_d + \tilde{G} + E) \\ &\quad + e^T \tilde{W}_M^T \tilde{W}_M \dot{e} + e^T \tilde{W}_M^T \dot{\tilde{W}}_M e \end{aligned} \quad (\text{A.2.31})$$

$$= -\dot{e}^T \hat{M}K_d \dot{e} - \dot{e}^T \hat{M}K_p e + \dot{e}^T (\tilde{M}\ddot{q}_d + \tilde{C}\dot{q}_d + \tilde{G} + E) + e^T \tilde{W}_M^T \tilde{W}_M \dot{e} + e^T \tilde{W}_M^T \dot{\tilde{W}}_M e \quad (\text{A.2.32})$$

where in the last step the skew symmetric property of $\dot{M} - 2C$ has been used.

Since the terms \ddot{q}_d and \dot{q}_d are bounded by choice all the terms in the expression

$$\tilde{M}\ddot{q}_d + \tilde{C}\dot{q}_d + \tilde{G} + E \quad (\text{A.2.33})$$

are bounded and it may be taken a maximum constant value of them as

$$\tilde{M}\ddot{q}_d + \tilde{C}\dot{q}_d + \tilde{G} + E \leq \theta \quad (\text{A.2.34})$$

Inserting this into (A.2.32) gives

$$\dot{V} \leq -\dot{e}^T \hat{M}K_d \dot{e} - \dot{e}^T \hat{M}K_p e + \dot{e}^T \theta + e^T \tilde{W}_M^T \tilde{W}_M \dot{e} + e^T \tilde{W}_M^T \dot{\tilde{W}}_M e \quad (\text{A.2.35})$$

$$= -\dot{e}^T \hat{M}K_d \dot{e} - \dot{e}^T (\hat{M}K_p - \tilde{W}_M^T \tilde{W}_M) e + \dot{e}^T \theta + e^T \tilde{W}_M^T \dot{\tilde{W}}_M e \quad (\text{A.2.36})$$

It is now looked at the last term. Here

$$\dot{\tilde{W}}_M = \dot{W} = \Delta W_M = -E_{AM} \bullet A_M \quad (\text{A.2.37})$$

from equation (A.2.18). The approximation error for the inertia matrix may be written as

$$E_{AM} = M - \hat{M} = M^* + E_M - \hat{M} = \tilde{M} + E_M \quad (\text{A.2.38})$$

This gives that the last term in equation (A.2.36) can be written as

$$e^T \tilde{W}_M^T \dot{\tilde{W}}_M e = e^T \tilde{W}_M^T \dot{\tilde{W}}_M e \quad (\text{A.2.39})$$

$$= e^T \tilde{W}_M^T (-E_{AM} \bullet A_M) e \quad (\text{A.2.40})$$

$$= -e^T \tilde{W}_M^T (\tilde{M} + E_M) \bullet A_M e \quad (\text{A.2.41})$$

$$= -e^T (\tilde{M} + E_M) \tilde{W}_M^T \bullet A_M e \quad (\text{A.2.42})$$

$$= -e^T (\tilde{M} + E_M) \tilde{M} e \quad (\text{A.2.43})$$

$$(\text{A.2.44})$$

Equation (A.2.36) then becomes

$$\dot{V} \leq -\dot{e}^T \hat{M} K_d \dot{e} - e^T (\tilde{M} + E_M) \tilde{M} e - \dot{e}^T (\hat{M} K_p - \tilde{W}_M^T \tilde{W}_M) e + \dot{e}^T \theta \quad (\text{A.2.45})$$

$$\leq -\dot{e}^T \hat{M} K_d \dot{e} - e^T (\tilde{M} + E_M) \tilde{M} e + \dot{e}^T (\hat{M} K_p - \tilde{W}_M^T \tilde{W}_M) e + \dot{e}^T \theta \quad (\text{A.2.46})$$

Now young's inequality may be used

$$\dot{e}^T (\hat{M} K_p - \tilde{W}_M^T \tilde{W}_M) e = (\hat{M} K_p - \tilde{W}_M^T \tilde{W}_M) \dot{e}^T I e \quad (\text{A.2.47})$$

$$\leq \frac{1}{2} (\hat{M} K_p - \tilde{W}_M^T \tilde{W}_M) (\dot{e}^T I \dot{e} + e^T I e) \quad (\text{A.2.48})$$

$$= \frac{1}{2} \dot{e}^T (\hat{M} K_p - \tilde{W}_M^T \tilde{W}_M) \dot{e} + \frac{1}{2} e^T (\hat{M} K_p - \tilde{W}_M^T \tilde{W}_M) e \quad (\text{A.2.49})$$

Inserting this into (A.2.46) gives

$$\dot{V} \leq -\dot{e}^T \hat{M} K_d \dot{e} - e^T (\tilde{M} + E_M) \tilde{M} e + \frac{1}{2} \dot{e}^T (\hat{M} K_p - \tilde{W}_M^T \tilde{W}_M) \dot{e} + \frac{1}{2} e^T (\hat{M} K_p - \tilde{W}_M^T \tilde{W}_M) e + \dot{e}^T \theta \quad (\text{A.2.50})$$

$$= -\dot{e}^T \left(\hat{M} (K_d - \frac{1}{2} K_p) + \frac{1}{2} \tilde{W}_M^T \tilde{W}_M \right) \dot{e} - e^T \left((\tilde{M} + E_M) \tilde{M} - \frac{1}{2} \hat{M} K_p + \frac{1}{2} \tilde{W}_M^T \tilde{W}_M \right) e + \dot{e}^T \theta \quad (\text{A.2.51})$$

$$\leq -\dot{e}^T \left(\hat{M} (K_d - \frac{1}{2} K_p) + \frac{1}{2} \tilde{W}_M^T \tilde{W}_M \right) \dot{e} - e^T \left((\tilde{M} + E_M^*) \tilde{M} - \frac{1}{2} \hat{M} K_p + \frac{1}{2} \tilde{W}_M^T \tilde{W}_M \right) e + \dot{e}^T \theta \quad (\text{A.2.52})$$

where the last line comes from (A.2.11).

For the second term actually to be negative then

$$(\tilde{M} + E_M^*)\tilde{M} - \frac{1}{2}\hat{M}K_p + \frac{1}{2}\tilde{W}_M^T\tilde{W}_M > 0 \quad (\text{A.2.53})$$

$$\Rightarrow -\frac{1}{2}\hat{M}K_p > -(\tilde{M} + E_M^*)\tilde{M} - \frac{1}{2}\tilde{W}_M^T\tilde{W}_M \quad (\text{A.2.54})$$

$$\Rightarrow -K_p > 2\hat{M}^{-1}\left(-(\tilde{M} + E_M^*)\tilde{M} - \frac{1}{2}\tilde{W}_M^T\tilde{W}_M\right) \quad (\text{A.2.55})$$

$$\Rightarrow K_p < 2\hat{M}^{-1}\left((\tilde{M} + E_M^*)\tilde{M} + \frac{1}{2}\tilde{W}_M^T\tilde{W}_M\right) \quad (\text{A.2.56})$$

For the first term actually to be negative then

$$\hat{M}(K_d - \frac{1}{2}K_p) + \frac{1}{2}\tilde{W}_M^T\tilde{W}_M > 0 \quad (\text{A.2.57})$$

$$\Rightarrow \hat{M}K_d - \frac{1}{2}\hat{M}K_p > -\frac{1}{2}\tilde{W}_M^T\tilde{W}_M \quad (\text{A.2.58})$$

By inserting from (A.2.56) it is found

$$\hat{M}K_d - \left((\tilde{M} + E_M^*)\tilde{M} + \frac{1}{2}\tilde{W}_M^T\tilde{W}_M\right) > -\frac{1}{2}\tilde{W}_M^T\tilde{W}_M \quad (\text{A.2.59})$$

$$\Rightarrow \hat{M}K_d > (\tilde{M} + E_M^*)\tilde{M} \quad (\text{A.2.60})$$

$$\Rightarrow K_d > \hat{M}^{-1}\left((\tilde{M} + E_M^*)\tilde{M}\right) \quad (\text{A.2.61})$$

Assuming that (A.2.56) and (A.2.61) are true equation (A.2.52) may be written as

$$\dot{V} \leq -\dot{e}^T\alpha\dot{e} - e^T\beta e + \dot{e}^T\theta \quad (\text{A.2.62})$$

where the the variables α and β are defined as

$$\begin{aligned} \alpha &\triangleq \hat{M}(K_d - \frac{1}{2}K_p) + \frac{1}{2}\tilde{W}_M^T\tilde{W}_M \\ \beta &\triangleq (\tilde{M} + E_M^*)\tilde{M} - \frac{1}{2}\hat{M}K_p + \frac{1}{2}\tilde{W}_M^T\tilde{W}_M \end{aligned}$$

By looking at \dot{V} as a storage function and the system (e, \dot{e}) with θ as input and \dot{e} as output it follows that this system will be state strictly passive according to definition 6.3 in [17]. By lemma 6.7 also in [17] the origin of the system will be 0-GAS.

Appendix B

ABB IRB140 Data Sheet

IRB 140 Industrial Robot

Main Applications

Arc welding
Assembly
Cleaning/Spraying
Machine tending
Material handling
Packing
Deburring



Small, Powerful and Fast

Compact, powerful IRB 140 industrial robot.

Six axis multipurpose robot that handles payload of 6 kg, with long reach (810 mm). The IRB 140 can be floor mounted, inverted or on the wall in any angle. Available as Standard, Foundry Plus 2, Clean Room and Wash versions, all mechanical arms completely IP67 protected, making IRB 140 easy to integrate in and suitable for a variety of applications. Uniquely extended radius of working area due to bend-back mechanism of upper arm, axis 1 rotation of 360 degrees even as wall mounted.

The compact, robust design with integrated cabling adds to overall flexibility. The Collision Detection option with full path retraction makes robot reliable and safe.

Using IRB 140T, cycle-times are considerably reduced where axis 1 and 2 predominantly are used.

Reductions between 15-20 % are possible using pure axis 1 and 2 movements. This faster versions is well suited for packing applications and guided operations together with PickMaster.

IRB 140 Foundry Plus 2 and Wash versions are suitable for operating in extreme foundry environments and other harsh environments with high requirements on corrosion resistance and tightness. In addition to the IP67 protection, excellent surface treatment makes the robot high pressure steam washable. Also available in white Clean Room ISO class 6 version, making it especially suited for environments with stringent cleanliness standards.

IRB 140

Specification

Robot versions	Handling capacity	Reach of 5th axis	Remarks
IRB 140/IRB 140T	6 kg	810 mm	
IRB 140F/IRB 140TF	6 kg	810 mm	Foundry Plus 2 Protection
IRB 140CR/IRB 140TCR	6 kg	810 mm	Clean Room
IRB 140W/IRB 140TW	6 kg	810 mm	SteamWash Protection
Supplementary load (on upper arm alt. wrist)			
on upper arm		1 kg	
on wrist		0.5 kg	
Number of axes			
Robot manipulator		6	
External devices		6	
Integrated signal supply	12 signals on upper arm		
Integrated air supply	Max. 8 bar on upper arm		
IRC5 Controller variants:	Single cabinet, Dual cabinet, Compact, Panel mounted		

Performance

Position repeatability 0.03 mm (average result from ISO test)

Axis movement	Axis	Working range
	1	360°
	2	200°
	3	280°
	4	Unlimited (400° default)
	5	240°
	6	Unlimited (800° default)

Max. TCP velocity 2.5 m/s

Max. TCP acceleration 20 m/s²

Acceleration time 0-1 m/s 0.15 sec

Velocity *)

Axis no.	IRB 140	IRB 140T
1	200°/s	250°/s
2	200°/s	250°/s
3	260°/s	260°/s
4	360°/s	360°/s
5	360°/s	360°/s
6	450°/s	450°/s

*) Max velocity is reduced at single phase power supply, e.g. Compact controller. Please, see the Product specification for further details.

Cycle time

5 kg Picking side	IRB 140	IRB 140T
cycle 25 x 300 x 25 mm	0.85s	0.77s

Electrical Connections

Supply voltage	200–600 V, 50/60 Hz
Rated power	
Transformer rating	4.5 kVA
Power consumption typically	0.4 kW

Physical

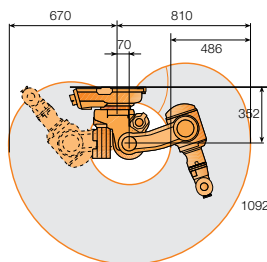
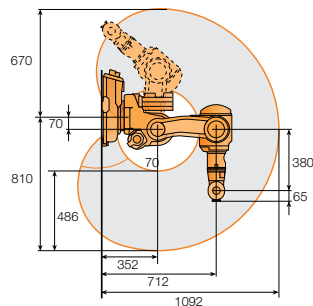
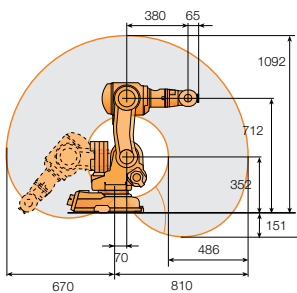
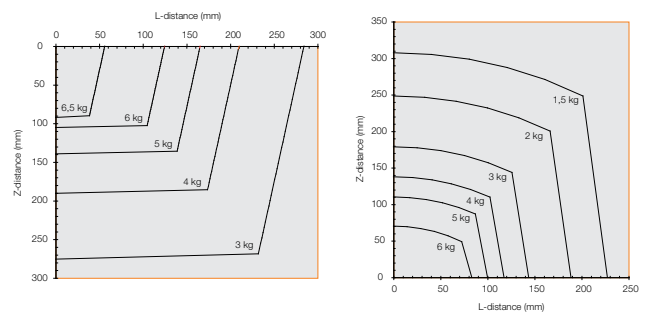
Robot mounting	Any angle
Dimensions	
Robot base	400 x 450 mm
Robot controller H x W x D	950 x 800 x 620 mm
Weight	
Robot manipulator	98 kg

Environment

Ambient temperature for	
Robot manipulator	5 – 45°C
Relative humidity	Max. 95%
Degree of protection,	
Manipulator	IP67
Options	
	Foundry Plus 2
	SteamWash (High pressure steam washable)
	Clean Room, class 6 (certified by IPA)
Noise level	Max. 70 dB (A)
Safety	
	Double circuits with supervision, emergency stops and safety functions, 3-position enable device
Emission	EMC/EMI-shielded

Data and dimensions may be changed without notice

Working range



Appendix C

Contents of Zip File

PDF of the report

Neural Network Files - +MRL

- `FA.m` - superclass definition of abstract function approximator class. From Sigurd Fjerdigen
- `RBFNN.m` - class `RBFNN`, inherits `FA`. Code originally from Sigurd Fjerdigen. The function `Calculation` changed to include activation function ratio calculations and pruning functions have been added.
- `ARBFNN.m` - class `ARBFNN`, inherits `FA`. Implement the growing networks `RANEKFs`. From Sigurd Fjerdigen.

Cross Function

- `CrossFunction.m` - do all the simulations concerning the cross function. Changeable variables to obtain different networks. Framework from Sigurd Fjerdigen.

2 dof manipulator

- **IniFile.m** - initialize all the networks and parameters for the manipulator. Gains used in the controller is also initialized here along with the pruning thresholds. Framework from Serge Gale.
- **Run.m** - runs the *IniFile.m*, start the simulation in *Simulink_RBFInvKinControl.mdl* and runs *Disp.m* when simulation done.
- **Disp.m** - display sizes of RBFNNs, tracking error, approximation errors, at what iteration the networks are pruned. Plots the desired trajectories and the actual trajectories. Comment/Uncomment to show different things.
- **Tau.m** - Inverse kinematic controller. Change between using the model and RBF networks. Framework from Serge Gale.
- **M.m**, **C.m** and **G.m** - Model of the 2 dof manipulator. From Stepan Pchelkin and Serge Gale.
- **M_hat.m**, **C_hat.m** and **G_hat.m** - estimate the manipulator model by using pruned RBFNNs. Framework from Serge Gale.
- **DynamicsOfTheManipulator.m** - Returns \dot{q} and \ddot{q} for the manipulator. Used in Simulink block *Dynamics*. Framework from Stepan Pchelkin and Serge Gale.
- **ForwardKinematic.m** - forwards kinematic to find end effector position. From Stepan Pchelkin and Serge Gale.
- **Rmse.m** - calculates the root mean square error of a error vector.
- **Simulink_RBFInvKinControl.mdl** - simulink diagram for the simulation. From Serge Gale and Stepan Pchelkin
- **DesiredTrajectory.mat** - desired trajectory file. From Stepan Pchelkin and Serge Gale.
- **time.mat** - the time belonging to the desired trajectory. From Stepan Pchelkin and Serge Gale.

ABB IRB140

- `Init.m` - initialize all the networks and parameters for the manipulator. Gains used in the controller is also initialized here along with the pruning thresholds. Displays the initial networks.
- `Run.m` - runs the `Init.m`, start the simulation in `NNinvKinCntrl_3dof_Sim.mdl` and runs `Disp.m` when simulation done.
- `Disp.m` - display sizes of RBFNNs, tracking error, approximation errors, at what iteration the networks are pruned. Plots the desired trajectories and the actual trajectories. Comment/Uncomment to show different things.
- `DesiredTrajectory.m` - generates the desired trajectory for each joint.
- `Tau.m` - Inverse kinematic controller. Change between using the model and RBF networks.
- `M.m`, `C.m` and `G.m` - Model of the ABB IRB140 manipulator. From Stepan Pchelkin.
- `M_hat.m`, `C_hat.m` and `G_hat.m` - estimate the manipulator model by using pruned RBFNNs.
- `DynamicsOfTheManipulator.m` - Returns \dot{q} and \ddot{q} for the manipulator. Used in Simulink block `Dynamics`. Framework from Stepan Pchelkin.
- `Rmse.m` - calculates the root mean square error of a error vector.
- `NNinvKinCntrl_3dof_Sim.mdl` - simulink diagram for the simulation.
- `K.mat` - coefficients for the parameters in the dynamic model of the manipulator. From Stepan Pchelkin.

Pruning of RBF Networks in Robot Manipulator Learning Control

Siri Vestheim*, Serge Gale*, Jan Tommy Gravdahl* and Sigurd Fjerdingen**

Abstract—Two simple and efficient pruning methods of RBF networks for use in a learning controller for robot manipulators will in this paper be proposed. They are called *Weight Magnitude Pruning* and *Node Output Pruning*.

Simulations with the two pruning methods in a learning inverse kinematic controller where the system dynamics are learned have been done. From the simulations it was found that implementing RBF networks to learn the system dynamics make the inverse kinematic controller more robust towards uncertainties and disturbances. Pruning these RBF networks further improved the performance in the case of modeling errors and reduced the computational cost.

I. INTRODUCTION

Over the last years robot manipulators have become a very important part of the industry and are used all over the world in factories and places with high risk involved for humans. Thus also the subject of how to control the manipulators in a best possible way has become an active research field. Previously model-based control for robotic manipulators has been found to give better performance than non-model based controllers. [1] The precision of the model-based regulators is closely linked to the accuracy of the dynamic model and the parameters in it. Finding a completely correct model is however not an easy task due to the robot manipulator itself and also the surrounding environment are containing a great deal of uncertainties and disturbances.

Controllers utilizing learning include methods coping well with the uncertainties and modeling errors. Out of the different learning schemes currently existing *Artificial Neural Networks* have proven to be well suited for storing the experienced knowledge of the system. They also have the ability to generalize from the knowledge on situations to new and unknown tasks similar to the already experienced ones.

Performance of a neural network is however very dependent on size of the network and the main problem with them is the *Curse of Dimensionality* [7]. To give an accurate approximation the networks have to be large enough. On the other hand if they should become too large the problem of *over-fitting* is likely to occur. Over-fitting is when there are too many hidden neurons and the network starts to fit the errors within the training data in addition to the underlying function. This gives poor generalization ability for the network.

Another major problem that follows with the curse of dimensionality is the computational cost of implementing

neural networks. When used in robot manipulator control it is absolutely necessary that the computations are fast enough so the outputs can be used in a regulator.

Radial Basis Function Neural Networks (RBFNN) is a type of artificial neural networks appropriate for use in robot manipulator learning control due certain properties like their known ability of universal approximation [13]. They are also reported to be computationally more efficient compared to multilayer perceptron networks in a number of control applications [14].

There are in general two ways to create a network with appropriate size, 1) *growing* and 2) *pruning*. Growing starts with a small or empty neural network and adds neurons until some threshold for the approximation error is met. The second method starts with an over dimensioned network and removes the weights and/or neurons that not are necessary for the network in order to be able to approximate. Pruning for artificial neural networks is based on the same concept as what naturally happens in the human brain when connections are pruned. For humans some synapses are cut to increase the brains ability to generalize from situation to situation when the humans are around the age of 20. The remaining connections in the brain then have their strength increased.[8]

Most of the developed pruning algorithms so far have been concerned with finding the optimal network with best possible approximation and generalization abilities. In order to be sure of not removing an incorrect neuron many pruning schemes have become very time demanding. Like the pruning method proposed in [16] which is based on complexity.

Some pruning schemes utilize two different networks in a performance check called *Cross Validation* to make sure that the removed unit is the correct one. [9] Both sets are pruned and the performance is found for each set separately before and after pruning. In [15] this performance check is used for a pruning scheme based on local sensitivity for the parameters in the network.

Sensitivity or saliency based pruning schemes are perhaps the most common group of pruning schemes. In this group the well known method *Optimal Brain Surgeon (OBS)* [6] can be found. This again is based on *Optimal Brain Damage (OBD)* [3] which proposed that the weight with the smallest saliency will generate the smallest error variation when removed. Both OBD and OBS start with a network that already has been trained to converge to a local minima and tries to minimize the cost function by setting one of the weights to zero. In OBD it was made an assumption that the hessian is diagonally dominant. However in [6] it was found that the hessian in fact was far from diagonally

*Department of Engineering Cybernetics, NTNU, NORWAY

*Department of Engineering Cybernetics, NTNU, NORWAY

*Department of Engineering Cybernetics, NTNU, NORWAY

**SINTEF ICT Applied Cybernetics, NORWAY

dominant and this resulted in OBD for some cases removing the wrong weights. Hence this assumption is not made in OBS and the saliency is found by solving the minimization problem with a Lagrangian multiplier. The weight with the lowest saliency then is pruned. From the same minimizing problem the optimal weight change is also found and this is used to update all the remaining weights. It is necessary to calculate the inverse Hessian for each weight that is pruned. Pruning is done until there are no more weights that can be removed without causing a large increase in the cost function.

All of the above mentioned pruning schemes find the optimal network. They are however not concerned with the computational cost the pruning algorithm needs or how computational demanding the final network is. For use in real-time control of a robot manipulator it is crucial that pruning happens fast since this will be done online while controlling the manipulator. In this setting it is probably better with a small network that perhaps not gives the best possible estimations but can give a satisfyingly approximation fast enough to be used in the controller.

Thus two simple pruning methods that fast removes many units and will give a network with satisfyingly approximation and generalization results by choosing an appropriate pruning threshold is here proposed. The first pruning method is based on Weight Powers Method in [5] and here enhanced into a scheme called Weight Magnitude Pruning. In the second proposed pruning method a definition from [2] is used. This is now put into a new setting in a Node Output Pruning scheme.

In the next section the two proposed pruning methods, Weight Magnitude Pruning and Node Output Pruning, are described. Then in section 3 a learning inverse kinematic controller with RBF networks that learns the system dynamics based on [4] and [12] is designed. This learning controller with the proposed pruning methods is then used in simulations for the first three joints of the ABB IRB140 manipulator in section 4.

Notation: RBFNN - radial basis function neural network, one training period - pruning and retraining once each, \hat{M} and other variables estimated by RBFNNs are written with a $\hat{}$ hat, network size is the number of hidden units.

II. PRUNING METHODS

The two pruning methods proposed in this paper are both developed for RBF networks with unity weights connecting the input layer and the hidden layer while the weights between the hidden layer and output layer are changeable. Due to the application these two pruning methods have been developed for RBF networks with one single output neuron is only taken into consideration. Thus removing a neuron and a weight will be equivalent. It should however not be any problem to extend the schemes to include networks with several output nodes.

With only one output neuron the final output of the network will be given as

$$F(x) = \sum_{i=1}^N w_i a_i(\|x - \mu_i\|) \quad (1)$$

where there are N units in the hidden layer, w_i is the weight between the output neuron and hidden unit i and $a_i(\|x - \mu_i\|)$ is the output of the activation function in hidden unit i to an input x . μ_i is the center for neuron i and $\|\cdot\|$ is the L^2 norm. The activation function is the *Gaussian* function given for node i as

$$a_i(\|x - \mu_i\|) = k_g e^{-\frac{\|x - \mu_i\|^2}{\sigma^2}} \quad (2)$$

where k_g is a positive constant, x is the input to the network and σ is the width of the activation function. Here the width is fixed the whole time and set to be the distance to the next unit.

From (1) it is possible to see that the output of the neural network is a sum of the contribution from each hidden neuron. This contribution depends in general on two factors, 1) the magnitude of the weights, and 2) the output of the activation functions. If one or both of them are zero or very small for a hidden unit this neuron may be removed from the network without affecting the output of the network much.

A. Weight Magnitude Pruning

This pruning method is based on a scheme in [5] and by looking at the magnitude of the weights the method finds the weights/units which are unnecessary for the network. A neuron that has a weight with small magnitude may be pruned. The novel idea of the weight magnitude pruning is using a certain percent of the L^2 norm of the weights as a threshold for pruning. This norm converges when the weights converge and also after some of the neurons in the network are removed it stays the same. Pruning some nodes result in the remaining ones having their weights increased in magnitude as to match the weights that were deleted. The pruning threshold is then found as

$$pW_{th} = \frac{\|W\| pW_{\%}}{100} \quad (3)$$

where $\|W\|$ is the L^2 norm of the weight vector and $pW_{\%}$ is the percent that the pruning threshold is of the weight norm. Now the the percent, $pW_{\%}$, is the only value to be chosen. Finding this should be less problem dependent than setting the threshold, pW_{th} , directly. Weights with a magnitude smaller than the threshold are then removed. This gives that there are several weights/neurons being pruned at the same time.

Weights change most at the beginning of the training phase and only little towards the end. The L^2 norm of the weights grows quickly to a certain value and stays approximately constant while the weights do some final smaller changes. Thus the weights not have to converge completely before pruning may begin.

It is not desired to remove all the weights in one period. A neuron which all the surrounding units have been pruned may

become much more important after the pruning then before. After pruning the remaining weights have their magnitude increased and thus the pruning threshold also must increase to remove more nodes. First time the network is pruned the specified threshold in percent, $pW\%$ is used. Next pruning the threshold is $2 * pW\%$, third time $3 * pW\%$ and so on until the threshold for pruning time N is $N * pW\%$. The number N is specified along with the percentage threshold.

Due to the localization principle [7] weights belonging to neurons in areas far from the input space will have their magnitude little increased. Hence these neurons are pruned with the weight magnitude pruning. In addition some neurons close to the inputs will have weights with small magnitude due to the area being too densely populated with nodes. These units are also pruned and the final network will obtain better generalization ability.

B. Neuron Output Pruning

The output from the activation function in the hidden neurons is the second factor which decides the contribution of a node to the final network output. Since the activation function here is the Gaussian function the size of the activation function output will be dependent on the distance from the inputs to the hidden neurons. Units in remote areas compared to the input space may thus be pruned. Here it is used an *activation function output ratio* which comes from [2]. This is given as

$$\sigma_i^j = \left| \frac{a_i(|j - \mu_i|)}{a_{max}} \right| \quad (4)$$

where j is the input, $a_i(|j - \mu_i|)$ is the activation function for neuron i , μ_i is the center of node i and a_{max} is the maximum activation function output for that given input. Activation function output ratio is here referred to as *node/neuron output* to shorten the writing. New in the neuron output pruning method is to sum up all the node output ratios for each neuron and remove those with a small sum. To find the nodes not close to the inputs a threshold found as a percent of the maximum output ratio sum is here proposed. The activation function output ratio sum is defined as

$$os_i = \sum_{j=1}^T \sigma_i^j \quad (5)$$

where T is the number of previous inputs and σ_i^j is the activation function output ratio for neuron i to input j . By specifying a threshold in percent the actual threshold, pO_{th} , will then be given as

$$pO_{th} = \frac{os_{max} * pO\%}{100} \quad (6)$$

where os_{max} is the highest output ratio sum and $pO\%$ is the pruning threshold percent.

Units in remote areas compared to where the inputs occur are here removed. This give a smaller network with good approximation ability due to all the active neurons are kept. However the generalization ability will not be much improved.

C. Mixed Method

The learning controller implemented in this paper has one RBF network to learn each of the elements in the dynamic matrices as done in [4]. This result in many smaller networks with different properties and thus would pruning them with different threshold or method would perhaps give better result than using the same for all of them.

III. CONTROL DESIGN

The control design is made for an ABB IRB140 manipulator which later will be used in simulations. This manipulator has 6 revolute joints where the three last are the wrist. Thus it is possible to do trajectory tracking by only controlling the first three joints as done in this paper.

Tracking error is found by using the sum squared error (SSE) function for all three joints while the network approximation error is found by the root mean square error (RMSE) function.

A. Manipulator Model

With friction and constant disturbance the dynamic equation for the robot manipulator can be written as

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) + F(\dot{q}) + d = \tau \quad (7)$$

where $M(q)$ is the inertia matrix, $C(q, \dot{q})$ is the centrifugal matrix, $G(q)$ is the gravitation matrix, $F(\dot{q})$ is the friction matrix and d is a constant disturbance. q, \dot{q}, \ddot{q} are joint position, joint velocity and joint acceleration respectively.

Friction and disturbance are implemented as $F = [1.2\dot{q}_1 \ 1.4\dot{q}_2 \ 0.8\dot{q}_3]^T$ and $d = [3 \ 5 \ 4]^T$.

For the situation where there are no friction or disturbance in the manipulator these terms are set to zero.

B. Controller and Desired Trajectory

An inverse kinematic controller from [12]:

$$\tau = M(q)(\ddot{q}_d + K_d(\dot{q}_d - \dot{q}) + K_p(q_d - q)) + C(q, \dot{q})\dot{q}_d + G(q) \quad (8)$$

and when the manipulator dynamics are implemented by RBF networks it can be written as

$$\tau = \hat{M}(q)(\ddot{q}_d + K_d(\dot{q}_d - \dot{q}) + K_p(q_d - q)) + \hat{C}(q, \dot{q})\dot{q}_d + \hat{G}(q) \quad (9)$$

Gain has been used: $K_p = \text{diag}(350, 450, 450)$ and $K_d = \text{diag}(2.5, 4.0, 2.5)$.

The desired trajectory is specified for each joint as

$$q_{1d}(t) = 0.5\sin(t) \quad (10)$$

$$q_{2d}(t) = 0.7\sin(t) \quad (11)$$

$$q_{3d}(t) = 0.3\sin(t) \quad (12)$$

C. Manipulator Model by RBF Networks

The changeable weights can be written in a vector as

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} \quad (13)$$

where w_1 is the weight between hidden unit 1 and the output unit, w_2 is the weight between hidden unit 2 and the output unit and N is the number of units in the hidden layer. In the beginning all the weights are initialized to zero.

Writing the activation functions in a vector gives

$$\mathbf{a}(\|x - \mu\|) = \begin{bmatrix} a_1(\|x - \mu_1\|) \\ a_2(\|x - \mu_2\|) \\ \vdots \\ a_N(\|x - \mu_N\|) \end{bmatrix} \quad (14)$$

To simplify the writing $\mathbf{a}(\|x - \mu\|)$ will be written as $\mathbf{a}(x)$ since the centres are fixed.

Now it is done as in [4] with one RBF network dedicated to each of the elements in the dynamic matrices.

The estimated inertia matrix, \hat{M} , can then be written as

$$\hat{M} = W_M^T \bullet A_M(x) \quad (15)$$

where W_M is the weight matrix defined as

$$W_M^T \triangleq \begin{bmatrix} \mathbf{w}_{m11}^T & \mathbf{w}_{m12}^T & \mathbf{w}_{m13}^T \\ \mathbf{w}_{m21}^T & \mathbf{w}_{m22}^T & \mathbf{w}_{m23}^T \\ \mathbf{w}_{m31}^T & \mathbf{w}_{m32}^T & \mathbf{w}_{m33}^T \end{bmatrix} \quad (16)$$

when the elements in the matrix such as \mathbf{w}_{m11} is a column vector as defined in (13). $A_M(x)$ is the activation function matrix for an input x , defined as

$$A_M(x) \triangleq \begin{bmatrix} \mathbf{a}_{m11}(x) & \mathbf{a}_{m12}(x) & \mathbf{a}_{m13}(x) \\ \mathbf{a}_{m12}(x) & \mathbf{a}_{m22}(x) & \mathbf{a}_{m23}(x) \\ \mathbf{a}_{m31}(x) & \mathbf{a}_{m32}(x) & \mathbf{a}_{m33}(x) \end{bmatrix} \quad (17)$$

where the elements are as in (14).

Finding the approximated C matrix follows the same pattern

$$\hat{C} = W_C^T \bullet A_C(x) \quad (18)$$

$$= \begin{bmatrix} \mathbf{w}_{c11}^T \mathbf{a}_{c11}(x) & \mathbf{w}_{c12}^T \mathbf{a}_{c12}(x) & \mathbf{w}_{c13}^T \mathbf{a}_{c13}(x) \\ \mathbf{w}_{c21}^T \mathbf{a}_{c21}(x) & \mathbf{w}_{c22}^T \mathbf{a}_{c22}(x) & \mathbf{w}_{c23}^T \mathbf{a}_{c23}(x) \\ \mathbf{w}_{c31}^T \mathbf{a}_{c31}(x) & \mathbf{w}_{c32}^T \mathbf{a}_{c32}(x) & \mathbf{w}_{c33}^T \mathbf{a}_{c33}(x) \end{bmatrix} \quad (19)$$

and so does the estimation of G

$$\hat{G} = W_G^T \bullet A_G(x) = \begin{bmatrix} \mathbf{w}_{g1}^T \mathbf{a}_{g1}(x) \\ \mathbf{w}_{g2}^T \mathbf{a}_{g2}(x) \\ \mathbf{w}_{g3}^T \mathbf{a}_{g3}(x) \end{bmatrix} \quad (20)$$

Since the inertia matrix is symmetric only 6 networks instead of 9 are needed to learn all the elements in $M(q)$. The similar elements in the inertia matrix are $m_{12} = m_{21}$, $m_{13} = m_{31}$ and $m_{23} = m_{32}$. It is also known that the

elements c_{33} and g_1 are zero and thus not needed to be estimated by RBF networks.

All the networks are spanned in the interval $[-\pi, \pi]$ with a distance of 0.75 between the nodes. In table I the initial networks and the inputs to the networks can be seen. Some of the joint velocities are added up and fed to the network as one input in order to reduce the number of inputs and then reduce the size of the networks.

Initial RBF Networks		
Network	Network Size	Inputs
\hat{m}_{11}	81	q_2, q_3
\hat{m}_{12}	81	q_2, q_3
\hat{m}_{13}	81	q_2, q_3
\hat{m}_{22}	9	q_3
\hat{m}_{23}	9	q_3
\hat{m}_{33}	9	1
\hat{c}_{11}	729	$q_2, q_3, (\dot{q}_2 + \dot{q}_3)$
\hat{c}_{12}	729	$q_2, q_3, (\dot{q}_2 + \dot{q}_2 + \dot{q}_3)$
\hat{c}_{13}	729	$q_2, q_3, (\dot{q}_2 + \dot{q}_2 + \dot{q}_3)$
\hat{c}_{21}	729	\dot{q}_1, q_2, q_3
\hat{c}_{22}	81	q_3, \dot{q}_3
\hat{c}_{23}	81	$q_3, (\dot{q}_2 + \dot{q}_3)$
\hat{c}_{31}	81	$(\dot{q}_1 + q_2), q_3$
\hat{c}_{32}	81	q_2, q_3
\hat{g}_1	81	q_2, q_3
\hat{g}_2	81	q_2, q_3
Sum	3672	-

TABLE I
INITIAL RBF NETWORKS

IV. SIMULATION

Simulations with the inverse kinematic controller for one case where there are no friction or disturbance in the manipulator and also for a case with friction and a constant disturbance present are done. For both cases the inverse kinematic controller is implemented with a model of the manipulator, the initial networks from table I without pruning and also the initial networks pruned by the proposed pruning methods. In addition to this simulations with the growing RBF networks called *Resource Allocation Networks Extended Kalman Filter (RANEKF)* [10] has been conducted to compare the performance from these growing networks to the performance of the pruned networks.

Simulations are done in Matlab and Simulink.

A. Pruning of Networks

Before the networks may be pruned it is crucial that they have experienced enough training. With weight magnitude pruning all of the weights have to be updated enough to avoid removing some wrong ones. Here the manipulator must have completed one half period of the first sinus wave before the networks can be pruned. For neuron output pruning it is necessary that the different areas of the input space all are visited since the pruning is done based on the distance from the units to the inputs. Thus the manipulator now had to complete one whole sinus wave, that is 2π simulation seconds and approximately 4400 iterations in

simulink. In addition to the simulation time criteria it is used that the rms estimation error over a moving window of 400 simulink iterations has to be smaller than 0.01 before a network may be pruned. The exception to this is the network \hat{g}_2 which can be pruned when the approximation error is less than 0.05 over the last 400 iterations.

All the networks are initially created as in table I and then pruned with the different methods. For the weight magnitude pruned networks it has been used a starting threshold as 5% which can grow 5 times. Threshold for the node output pruning has been taken as 30%. Due to the threshold being specified as a percent the same threshold for all the networks is possible to use even if they are quite different. Looking at the networks more separately however will make it possible to obtain better result. Thus a mixed method where it has been used different pruning for the estimated dynamic matrices was implemented. Networks belonging to \hat{M} was pruned with 7% weight based pruning while it was used 30% node output pruning for the \hat{C} networks and 5% weight magnitude pruning for the two networks belonging to \hat{G} .

B. Obtained Networks After Pruning

In table II the final networks after pruning as described in the last section can be seen. Here the final number of hidden units in the different networks and the approximation error for the last 300 iterations of the simulation for each network are shown. The approximation error is only shown for the last part of the simulation in order to show the performance of the networks after they are pruned and retrained. From this table it is visible that pruning the networks with the proposed method either separately or mixed for the different dynamic matrices reduce the number of hidden units a lot. Weight magnitude pruning obtained the networks with fewest hidden units in total while the mixed methods and the node output pruning almost had the same number.

C. Results for the Case Without Friction and Disturbance

In table III the simulation results from implementing the system dynamics in the inverse kinematic controller in different ways are visible. Here the method, the total number of hidden units for all the networks, the tracking error for the whole simulation and the approximation error for all the networks over last 300 iterations of the simulation are visible. The latter is shown for both a simulation with time 6π seconds and a simulation of 20π seconds.

As can be seen from this table implementing the correct model of the manipulator in the inverse kinematic controller gives the best tracking result. This would be as expected. It can also be seen that not pruning the networks give better tracking result than pruning them. Out of the pruned network the mixed pruning method gives the smallest tracking error. RANEKFs have the largest tracking error which probably is

Resulting Networks After Pruning						
NN	Weight Magn.		Node Output		Mixed Method	
	NN Size	Approx. Error [RMSE]	NN Size	Approx. Error [RMSE]	NN Size	Approx. Error [RMSE]
\hat{m}_{11}	6	0.0067127	5	0.0061364	4	0.0054748
\hat{m}_{12}	6	0.0005721	5	0.00050466	4	0.00034923
\hat{m}_{13}	2	0.001048	5	0.00075247	2	0.0010479
\hat{m}_{22}	3	0.004117	2	0.0065574	3	0.0041168
\hat{m}_{23}	3	0.001939	2	0.0028382	3	0.0019389
\hat{m}_{33}	2	≈ 0	2	0	2	0
\hat{c}_{11}	3	0.0083142	17	0.0016613	17	0.0016601
\hat{c}_{12}	13	0.0023642	22	0.0037684	22	0.0037649
\hat{c}_{13}	8	0.0072465	22	0.00073191	22	0.00073047
\hat{c}_{21}	3	0.0072956	11	0.00084954	11	0.00084958
\hat{c}_{22}	2	0.0061949	4	0.0012086	4	0.0012063
\hat{c}_{23}	3	0.0006785	7	0.0010091	7	0.0010054
\hat{c}_{31}	2	0.0069534	6	0.0043594	6	0.0043596
\hat{c}_{32}	2	0.0014019	6	0.0011061	6	0.0011059
\hat{g}_1	10	0.23334	5	0.23923	10	0.23334
\hat{g}_2	6	0.0021934	5	0.0095059	6	0.002195
Sum	74	0.29037	126	0.28021	129	0.26314

TABLE II
RESULTING NETWORKS AFTER PRUNING

due to the poorer approximation ability at the beginning of the simulation when enough neurons has not been added.

Not pruning the networks also gave smaller approximation error in total for all the networks than the pruned ones did. Again the mixed method gives the best result from the pruned networks. Here RANEKFs are far superior compared to the other RBF networks with a much smaller estimation error. However the approximation error increases when the simulation length is increased for the RANEKFs. Also the networks did not stop to add neurons and after 20π simulation time the total number of hidden neurons in the RANEKFs was 147. Thus it seems like the RANEKFs keep growing and start to overfit the simulation data as the time goes. All of the other networks had the same sizes after the longer simulation and their estimations all improved.

It can also be mentioned that the actual time it took to complete the simulation was much longer when RANEKFs were implemented. It then took 7.20 minutes while the simulation with weight based pruning took 4.10 minutes to complete. Both node output and mixed methods pruning used 4.20 minutes for the whole simulation. Not pruning the networks gave an actual time of 4.50 minutes.

In figure 1 a plot of the desired trajectories and actual trajectories for each joint when it is used a learning inverse kinematic controller with 5% weight magnitude pruning can be seen. Here it is visible that the obtained tracking is very accurate.

D. Results for the Case With Friction and Disturbance

Now a simple model of friction and a constant disturbance are included in the manipulator. These dynamics are not added to the model of the manipulator which is used to train the networks and also implemented in the inverse kinematic controller. The networks ability to generalize will thus be

Trajectory Tracking and Network Approximation Errors				
Method	Sum of Hidden Units	Tracking Error [SSE]	Approx. Err [RMSE] 6π	Approx. Err [RMSE] 20π
Correct Model	-	0.031897	-	-
Unpruned	3672	0.032	0.26303	0.24867
Weight Magnitude	74	0.03201	0.29037	0.28429
Neuron Output	126	0.032002	0.28021	0.2657
Mixed Methods	129	0.032001	0.26314	0.25074
RANEKF	96	0.032094	0.024542	0.058592

TABLE III

TRAJECTORY TRACKING AND APPROXIMATION ERRORS

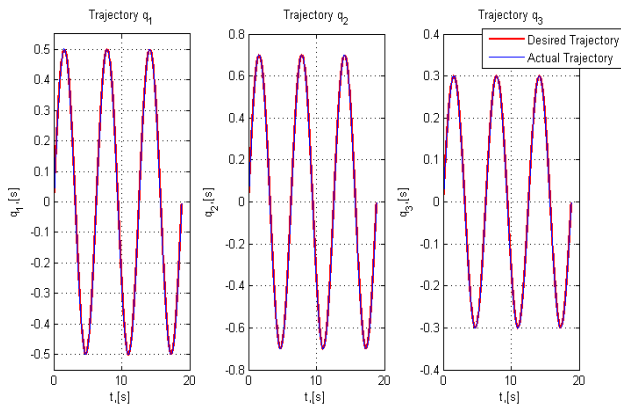


Fig. 1. Trajectory all 3 Joints Using Weight Pruned RBFNN

tested.

Results from simulations that lasted 6π seconds and included friction and disturbance are shown in table IV. This table shows the method that is used to implement the system dynamics in the controller, the total number of hidden units in the final networks, the tracking error for all three joints for the whole simulation and also the tracking error for only the last 2π simulation time. During the whole simulation the manipulator completes 3 whole sinus waves and it is found the tracking error of only the last sinus wave to see how the networks performs in tracking after being pruned.

For the networks pruned with node output and for the obtained networks from mixed methods pruning the size of all the networks were not the same. The two networks for \hat{c}_{12} and \hat{c}_{13} had one unit less than they had after pruning in the control of a manipulator without friction and disturbance. This is due to the change of inputs and thus the activation function ratios are no longer the same as in the previous situation.

Best tracking for both the whole simulation and the last part only is obtained with the learning controller with mixed methods pruned RBF networks. All the pruned networks give more accurate tracking than the other methods. Worst tracking for the whole simulation come from when the RANKEFs are implemented. In the final wave of the trajectory they do however obtain better tracking than the now incorrect model. RANEKFs are much smaller than for the case without friction and disturbance. This is probably due to it here being

more difficult to overfit the training data.

Disturbance and Friction Trajectory Tracking and Network Approximation Errors			
Method	Sum of Hidden Units	Tracking Error Whole Sim. [SSE]	Track. Error Last 2π Time [SSE]
Incorrect Model	-	0.052887	0.0072445
Unpruned Networks	3672	0.052977	0.0072385
Weight Magnitude	74	0.052898	0.0072129
Neuron Output	124	0.052975	0.0072379
Mixed Methods	127	0.052563	0.0071018
RANEKF	78	0.053116	0.007243

TABLE IV

DISTURBANCE AND FRICTION - TRAJECTORY TRACKING AND APPROXIMATION ERRORS

A plot of the desired joint trajectories together with the actual joint trajectories for the case with friction and disturbance can be seen in figure 2. Here mixed method pruning of the RBF networks in the learning inverse kinematic controller has been used. Also in this case the tracking result is very good.

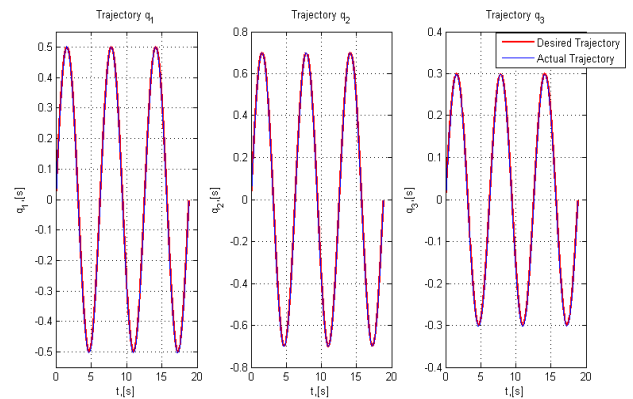


Fig. 2. Friction and Disturbance - Trajectory all 3 Joints Using Mixed Methods Pruning

V. CONCLUDING REMARKS

A. Conclusion

In this paper two simple and efficient pruning methods of RBF network for implementation in a learning controller for robot manipulators have been presented. Simulations with the pruning methods in a learning inverse kinematic controller where the system dynamics are learned have been done. Implementing this learning inverse kinematic controller instead of the normal model-based version made the system more robust towards uncertainties like friction and disturbance. If the learning was implemented with pruning schemes it was found that the generalization ability was improved and the computational cost reduced.

B. Future Work

The learning here is found to be ill-posed since the training data contain too little information about the desired solution. [7] Networks for the \hat{C} matrix which were dependent on several variable, q and \dot{q} , had more ill-posed learning than the networks depending on fewer variables.

A solution to the ill-posed problem is regularization and hence to see if the learning can be made more well-posed this should be implemented. Especially if it should be included control of the remaining 3 joints of the manipulator which would give system dynamics dependent on even more variables and hence an even more ill-posed problem.

STABILITY FOR THE LEARNING INVERSE KINEMATIC CONTROLLER

C. RBF Notation

From RBF theory an optimal network can approximate any given function with an arbitrary small approximation error [13]. For a matrix, F , the optimal estimation may be given as

$$F = F^* + E = W^{*T} \bullet A(x) + E \quad (21)$$

where F^* is the optimal estimation, E is the network approximation error, W^* are the optimal weights and $A(x)$ is the activation function matrix. With optimal weights it is normally assumed that $E < E^*$ with $E^* > 0$ and very small may be taken as a bound on the optimal approximation error.

The difference from the actual network estimation and optimal estimation is defined as

$$\tilde{F} \triangleq F^* - \hat{F} \quad (22)$$

For training the networks the gradient descent (delta rule) algorithm is used. The update of \hat{W} is then given as

$$\dot{\hat{W}} = -E_A \bullet A(x) = \begin{bmatrix} -e_{A11}a_{11}(x) & \cdots & -e_{A1n}a_{1n}(x) \\ \vdots & \ddots & \vdots \\ -e_{An1}a_{n1}(x) & \cdots & -e_{Ann}a_{nn}(x) \end{bmatrix} \quad (23)$$

where E_A is the approximation error matrix for the networks to a given input x and n is the number of joints.

D. Error Equation

Defines the tracking error variable as

$$e \triangleq q_d - q \quad (24)$$

where q_d is the desired trajectory and q is the actual joint position.

An error equation can then be found starting with the dynamic equation

$$M\ddot{q} + C\dot{q} + G = \tau \Rightarrow \quad (25)$$

$$-M\ddot{e} + M^*\dot{q}_d + E_M\ddot{q}_d - C\dot{e} + C^*\dot{q}_d + E_C\dot{q}_d + G^* + E_G = \tau \quad (26)$$

where it has been used (24) and (21). Now a new approximation error is defined as

$$E \triangleq E_M\ddot{q}_d + E_C\dot{q}_d + E_G \quad (27)$$

Using (27), (9) and (22) it is obtained

$$\tilde{M}\ddot{q}_d + \tilde{C}\dot{q}_d + \tilde{G} + E = M\ddot{e} + C\dot{e} + \hat{M}K_d\dot{e} + \hat{M}K_p e \quad (28)$$

E. Lyapunov Function Candidate

It is started with the following Lyapunov Function Candidate

$$V(e, \dot{e}) = \frac{1}{2}\dot{e}^T M(q)\dot{e} + \frac{1}{2}e^T \tilde{W}_M^T \tilde{W}_M e \quad (29)$$

which has $V(0,0) = 0$, is positive definite and radially unbounded when it is assumed that $M(q) = M(q)^T$ is positive definite.

The derivative is

$$\dot{V} = \dot{e}^T M\ddot{e} + \frac{1}{2}\dot{e}^T \dot{M}\dot{e} + e^T \tilde{W}_M^T \tilde{W}_M \dot{e} + e^T \tilde{W}_M^T \dot{\tilde{W}}_M e \quad (30)$$

Inserting for $M\ddot{e}$ with (28) and using the skew symmetric property of $\dot{M} - 2C$ give

$$\dot{V} = -\dot{e}^T \hat{M}K_d\dot{e} - \dot{e}^T (\hat{M}K_p - \tilde{W}_M^T \tilde{W}_M) e + \dot{e}^T (\tilde{M}\ddot{q}_d + \tilde{C}\dot{q}_d + \tilde{G} + E) + e^T \tilde{W}_M^T \dot{\tilde{W}}_M e \quad (31)$$

From [7] the weights are guaranteed to converge with the training algorithm used and thus the weights are bounded. The activation functions are bounded from the localization principle and hence is \hat{F} and F^* are for all the matrices bounded. By choice \ddot{q}_d and \dot{q}_d may be bounded. Since all the terms are bounded it follows that an upper bound on the following expression may be taken as

$$\tilde{M}\ddot{q}_d + \tilde{C}\dot{q}_d + \tilde{G} + E \leq \theta \quad (32)$$

Using the fact that $\dot{\tilde{W}} = \dot{\hat{W}}$ and inserting with (23) and (32) in (31) give

$$\dot{V} = -\dot{e}^T \hat{M}K_d\dot{e} - \dot{e}^T (\hat{M}K_p - \tilde{W}_M^T \tilde{W}_M) e + \dot{e}^T \theta + e^T \tilde{W}_M^T (-E_{AM} \bullet A_M(x)) e \quad (33)$$

where the last term may be rewritten as $E_{AM} = M - \hat{M}$, (21) and (22)

$$-e^T (\tilde{M} + E_M) \tilde{W}_M^T \bullet A_M e = -e^T (\tilde{M} + E_M) \tilde{M} e \quad (34)$$

Inserting this back into (33)

$$\dot{V} = -\dot{e}^T \hat{M}K_d\dot{e} - e^T (\tilde{M} + E_M) \tilde{M} e - \dot{e}^T (\hat{M}K_p - \tilde{W}_M^T \tilde{W}_M) e + \dot{e}^T \theta \quad (35)$$

$$\leq -\dot{e}^T \hat{M}K_d\dot{e} - e^T (\tilde{M} + E_M^*) \tilde{M} e + \dot{e}^T (\hat{M}K_p - \tilde{W}_M^T \tilde{W}_M) e + \dot{e}^T \theta \quad (36)$$

$$(37)$$

Now young's inequality is used on the third term and it is obtained

$$\begin{aligned} \dot{V} &\leq -\dot{e}^T \hat{M} K_d \dot{e} - e^T (\tilde{M} + E_M^*) \tilde{M} \dot{e} + \dot{e}^T \theta \\ &\quad + \frac{1}{2} \dot{e}^T (\hat{M} K_p - \tilde{W}_M^T \tilde{W}_M) \dot{e} + \frac{1}{2} e^T (\hat{M} K_p - \tilde{W}_M^T \tilde{W}_M) e \end{aligned} \quad (38)$$

$$\begin{aligned} &= -\dot{e}^T \left(\hat{M} (K_d - \frac{1}{2} K_p) + \frac{1}{2} \tilde{W}_M^T \tilde{W}_M \right) \dot{e} - \\ &\quad e^T \left((\tilde{M} + E_M^*) \tilde{M} - \frac{1}{2} \hat{M} K_p + \frac{1}{2} \tilde{W}_M^T \tilde{W}_M \right) e + \dot{e}^T \theta \end{aligned} \quad (39)$$

For the second term actually to be negative then

$$K_p < 2\hat{M}^{-1} \left((\tilde{M} + E_M^*) \tilde{M} + \frac{1}{2} \tilde{W}_M^T \tilde{W}_M \right) \quad (40)$$

For the first term actually to be negative then by inserting with (40)

$$K_d > \hat{M}^{-1} \left((\tilde{M} + E_M^*) \tilde{M} \right) \quad (41)$$

Assuming that (40) and (41) are true equation (39) may be written as

$$\dot{V} \leq -\dot{e}^T \alpha \dot{e} - e^T \beta e + \dot{e}^T \theta \quad (42)$$

By looking at \dot{V} as a storage function and the system (e, \dot{e}) with θ as input and \dot{e} as output it follows that the system will be state strictly passive according to definition 6.3 in [11]. By lemma 6.7 also in [11] the origin of the system will be 0-GAS.

ACKNOWLEDGMENT

Thanks to Stepan Pchelkin for a model of the three first joints of the ABB IRBE140 manipulator.

REFERENCES

- [1] Chae H. An, Christopher G. Atkeson, and John M. Hollerbach. Model-based control of a direct drive arm, part 2: Control. *1988 IEEE International Conference on Robotics and Automation*, 3.
- [2] Somwang Arisariyawong and Siam Charoenseang. Dynamic self-organized learning for optimizing the complexity growth of radial basis function neural networks. *IEEE International Conference on Industrial Technology, 2002. IEEE ICIT '02.*, 1.
- [3] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal brain damage. *Proceedings of the Neural Information Processing Systems*.
- [4] S.S. Ge, T.H. Lee, and C.J. Harris. *Adaptive Neural Network Control of Robotic Manipulators*. World Scientific Publishing Co, Singapore, 1998.
- [5] Masafumi Hagiwara. Removal of hidden units and weights for back propagation network. *Proceedings of 1993 International Joint Conference on Neural Networks, 1993. IJCNN '93-Nagoya.*, 1.
- [6] Babak Hassibi, David G. Stork, and Gregory J. Wolff. Optimal brain surgeon and general network pruning. *IEEE International Conference on Neural Networks 1993*.
- [7] Simon Haykin. *Neural Networks – A Comprehensive Foundation*. Pearson - Prentice Hall, ninth reprint 2005 edition, 1999.
- [8] Ralph E. Hoffman and Thomas H. McGlashan. Neural network models of schizophrenia. *The Neuroscientist*, 2001.
- [9] Thuan Q. Huynh and Rudy Setiono. Effective nn pruning using cross-validation. *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005. IJCNN '05.*, 2.
- [10] Visakan Kadiramanathan and Mahesan Niranjan. A function estimation approach to sequential learning with neural networks. *Neural Computation*, 5.
- [11] Hassan K. Khalil. *Nonlinear systems*. Prentice-Hall, third, international edition, 2000.

- [12] M. Vidyasagar Mark W. Spong, Seth Hutchinson. *Robot Modeling and Control*. John Wiley and Sons, INC.
- [13] J. Park and I. W. Sandberg. Universal approximation using radial-basis-function networks. *Neural Comput.*, 3(2):246–257, 1991.
- [14] Jung-Wok Park, R.G. Harley, and G.K. Venayagamoorthy. Comparison of mlp and rbf neural networks using deviation signals for on-line identification of a synchronous generator. *IEEE Power Engineering Society Winter Meeting, 2002*, 1.
- [15] Devin Sabo and Xiao-Hua Yu. A new pruning algorithm for neural network dimension analysis. *IEEE International Joint Conference on Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence)*.
- [16] Zhaozhao Zhang and Junfei Qiao. A novel pruning algorithm for feedforward neural network based on neural complexity. *2010 International Conference on Intelligent Control and Information Processing (ICICIP)*.